

# Picalo Manual For Advanced Users and Programmers

Conan C. Albrecht, PhD

December 17, 2009



# Contents

<b>1</b>	<b>Introduction</b>	<b>10</b>
1.1	Picalo Architecture . . . . .	10
<b>2</b>	<b>Picalo Tables</b>	<b>12</b>
2.1	Creating Tables . . . . .	13
2.2	Columns and Headings . . . . .	15
2.3	Records . . . . .	20
2.4	Viewing Tables . . . . .	25
<b>3</b>	<b>Data Types</b>	<b>26</b>
3.1	Table . . . . .	26
3.2	String . . . . .	27
3.3	Number Types . . . . .	27
3.4	DateTime . . . . .	28
3.5	None . . . . .	29
3.6	Error . . . . .	29
3.7	Other Types . . . . .	30
<b>4</b>	<b>Databases</b>	<b>31</b>
4.1	Creating the Connection . . . . .	31
4.2	Querying Data . . . . .	32
4.3	Modifying Data . . . . .	33
<b>5</b>	<b>Scripting</b>	<b>35</b>
5.1	The Shell . . . . .	36
5.2	Script Files . . . . .	36
5.3	Running Scripts . . . . .	37
5.4	Ideas for Successful Scripts . . . . .	37

5.5	The Python Language . . . . .	38
<b>6</b>	<b>Picalo Reference</b>	<b>40</b>
<b>7</b>	<b>Core</b>	<b>41</b>
7.1	boolean . . . . .	41
7.2	check_valid_table . . . . .	41
7.3	clear_progress . . . . .	42
7.4	count . . . . .	42
7.5	currency.adjusted . . . . .	42
7.6	currency.as_tuple . . . . .	43
7.7	currency.canonical . . . . .	43
7.8	currency.compare . . . . .	43
7.9	currency.compare_signal . . . . .	44
7.10	currency.compare_total . . . . .	44
7.11	currency.compare_total_mag . . . . .	44
7.12	currency.conjugate . . . . .	45
7.13	currency.copy_abs . . . . .	45
7.14	currency.copy_negate . . . . .	45
7.15	currency.copy_sign . . . . .	45
7.16	currency.exp . . . . .	46
7.17	currency.fma . . . . .	46
7.18	currency.is_canonical . . . . .	46
7.19	currency.is_finite . . . . .	47
7.20	currency.is_infinite . . . . .	47
7.21	currency.is_nan . . . . .	47
7.22	currency.is_normal . . . . .	47
7.23	currency.is_qnan . . . . .	48
7.24	currency.is_signed . . . . .	48
7.25	currency.is_snan . . . . .	48
7.26	currency.is_subnormal . . . . .	48
7.27	currency.is_zero . . . . .	48
7.28	currency.ln . . . . .	49
7.29	currency.log10 . . . . .	49
7.30	currency.logb . . . . .	49
7.31	currency.logical_and . . . . .	50
7.32	currency.logical_invert . . . . .	50
7.33	currency.logical_or . . . . .	50

7.34	currency.logical_xor . . . . .	51
7.35	currency.max . . . . .	51
7.36	currency.max_mag . . . . .	51
7.37	currency.min . . . . .	52
7.38	currency.min_mag . . . . .	52
7.39	currency.next_minus . . . . .	52
7.40	currency.next_plus . . . . .	53
7.41	currency.next_toward . . . . .	53
7.42	currency.normalize . . . . .	53
7.43	currency.number_class . . . . .	54
7.44	currency.quantize . . . . .	54
7.45	currency.radix . . . . .	54
7.46	currency.remainder_near . . . . .	55
7.47	currency.rotate . . . . .	55
7.48	currency.same_quantum . . . . .	55
7.49	currency.scaleb . . . . .	56
7.50	currency.shift . . . . .	56
7.51	currency.sqrt . . . . .	56
7.52	currency.to_eng_string . . . . .	57
7.53	currency.to_integral . . . . .	57
7.54	currency.to_integral_exact . . . . .	57
7.55	currency.to_integral_value . . . . .	58
7.56	DateDelta . . . . .	58
7.57	DateFormat . . . . .	59
7.58	DateTimeFormat . . . . .	60
7.59	error . . . . .	60
7.60	error.get_exception . . . . .	61
7.61	error.get_message . . . . .	61
7.62	error.get_previous . . . . .	61
7.63	load . . . . .	62
7.64	load_csv . . . . .	62
7.65	load_delimited . . . . .	63
7.66	load_excel . . . . .	64
7.67	load_fixed . . . . .	65
7.68	load_tsv . . . . .	67
7.69	max . . . . .	68
7.70	mean . . . . .	68
7.71	min . . . . .	69

7.72	number.adjusted	69
7.73	number.as_tuple	69
7.74	number.canonical	69
7.75	number.compare	70
7.76	number.compare_signal	70
7.77	number.compare_total	70
7.78	number.compare_total_mag	71
7.79	number.conjugate	71
7.80	number.copy_abs	71
7.81	number.copy_negate	71
7.82	number.copy_sign	72
7.83	number.exp	72
7.84	number.fma	72
7.85	number.is_canonical	73
7.86	number.is_finite	73
7.87	number.is_infinite	73
7.88	number.is_nan	73
7.89	number.is_normal	74
7.90	number.is_qnan	74
7.91	number.is_signed	74
7.92	number.is_snan	74
7.93	number.is_subnormal	75
7.94	number.is_zero	75
7.95	number.ln	75
7.96	number.log10	75
7.97	number.logb	76
7.98	number.logical_and	76
7.99	number.logical_invert	76
7.100	number.logical_or	77
7.101	number.logical_xor	77
7.102	number.max	77
7.103	number.max_mag	78
7.104	number.min	78
7.105	number.min_mag	78
7.106	number.next_minus	79
7.107	number.next_plus	79
7.108	number.next_toward	79
7.109	number.normalize	80

7.110	number.number_class . . . . .	80
7.111	number.quantize . . . . .	80
7.112	number.radix . . . . .	81
7.113	number.remainder_near . . . . .	81
7.114	number.rotate . . . . .	81
7.115	number.same_quantum . . . . .	82
7.116	number.scaleb . . . . .	82
7.117	number.shift . . . . .	82
7.118	number.sqrt . . . . .	83
7.119	number.to_eng_string . . . . .	83
7.120	number.to_integral . . . . .	83
7.121	number.to_integral_exact . . . . .	84
7.122	number.to_integral_value . . . . .	84
7.123	save_csv . . . . .	84
7.124	save_delimited . . . . .	85
7.125	save_excel . . . . .	87
7.126	save_fixed . . . . .	87
7.127	save_tsv . . . . .	88
7.128	save_xml . . . . .	89
7.129	show_progress . . . . .	90
7.130	stdev . . . . .	91
7.131	sum . . . . .	92
7.132	Table . . . . .	92
7.133	Table.append . . . . .	94
7.134	Table.append_calculated . . . . .	94
7.135	Table.append_calculated_static . . . . .	95
7.136	Table.append_column . . . . .	96
7.137	Table.clear_filter . . . . .	97
7.138	Table.column . . . . .	97
7.139	Table.column_count . . . . .	97
7.140	Table.delete_column . . . . .	98
7.141	Table.deref_column . . . . .	98
7.142	Table.extend . . . . .	99
7.143	Table.filter . . . . .	99
7.144	Table.find . . . . .	100
7.145	Table.get_column_names . . . . .	100
7.146	Table.get_columns . . . . .	101
7.147	Table.get_filter_expression . . . . .	101

7.148	Table.guess_types	101
7.149	Table.index	102
7.150	Table.insert	102
7.151	Table.insert_calculated	103
7.152	Table.insert_calculated_static	104
7.153	Table.insert_column	105
7.154	Table.is_changed	105
7.155	Table.is_filtered	106
7.156	Table.is_readonly	106
7.157	Table.iterator	106
7.158	Table.move_column	107
7.159	Table.prettyprint	107
7.160	Table.record	109
7.161	Table.record_count	110
7.162	Table.reorder_columns	110
7.163	Table.save	110
7.164	Table.save_csv	111
7.165	Table.save_delimited	112
7.166	Table.save_excel	113
7.167	Table.save_fixed	113
7.168	Table.save_tsv	114
7.169	Table.save_xml	115
7.170	Table.set_changed	116
7.171	Table.set_format	116
7.172	Table.set_name	117
7.173	Table.set_readonly	117
7.174	Table.set_type	117
7.175	Table.sort	118
7.176	Table.structure	118
7.177	Table.view	119
7.178	TableArray	119
7.179	TableArray.append	120
7.180	TableArray.append_calculated	120
7.181	TableArray.append_calculated_static	121
7.182	TableArray.append_column	121
7.183	TableArray.clear_filter	122
7.184	TableArray.column	122
7.185	TableArray.column_count	122



7.186	TableArray.combine . . . . .	122
7.187	TableArray.delete_column . . . . .	123
7.188	TableArray.filter . . . . .	123
7.189	TableArray.get_column_names . . . . .	123
7.190	TableArray.get_columns . . . . .	123
7.191	TableArray.get_filter_expression . . . . .	124
7.192	TableArray.guess_types . . . . .	124
7.193	TableArray.insert_calculated . . . . .	124
7.194	TableArray.insert_calculated_static . . . . .	125
7.195	TableArray.insert_column . . . . .	125
7.196	TableArray.is_changed . . . . .	126
7.197	TableArray.is_filtered . . . . .	126
7.198	TableArray.is_readonly . . . . .	126
7.199	TableArray.move_column . . . . .	126
7.200	TableArray.save . . . . .	127
7.201	TableArray.save_csv . . . . .	127
7.202	TableArray.save_delimited . . . . .	128
7.203	TableArray.save_fixed . . . . .	129
7.204	TableArray.save_tsv . . . . .	130
7.205	TableArray.save_xml . . . . .	130
7.206	TableArray.set_changed . . . . .	131
7.207	TableArray.set_format . . . . .	132
7.208	TableArray.set_name . . . . .	132
7.209	TableArray.set_readonly . . . . .	132
7.210	TableArray.set_type . . . . .	133
7.211	TableArray.structure . . . . .	133
7.212	TableArray.view . . . . .	133
7.213	TableList . . . . .	134
7.214	TableList.append . . . . .	134
7.215	TableList.append_calculated . . . . .	135
7.216	TableList.append_calculated_static . . . . .	135
7.217	TableList.append_column . . . . .	136
7.218	TableList.clear_filter . . . . .	136
7.219	TableList.column . . . . .	136
7.220	TableList.column_count . . . . .	137
7.221	TableList.delete_column . . . . .	137
7.222	TableList.filter . . . . .	137
7.223	TableList.get_column_names . . . . .	137

7.224	TableList.get_columns . . . . .	138
7.225	TableList.guess_types . . . . .	138
7.226	TableList.insert_calculated . . . . .	138
7.227	TableList.insert_calculated_static . . . . .	139
7.228	TableList.insert_column . . . . .	139
7.229	TableList.is_changed . . . . .	140
7.230	TableList.is_filtered . . . . .	140
7.231	TableList.is_readonly . . . . .	140
7.232	TableList.move_column . . . . .	140
7.233	TableList.save . . . . .	141
7.234	TableList.save_csv . . . . .	141
7.235	TableList.save_delimited . . . . .	142
7.236	TableList.save_fixed . . . . .	143
7.237	TableList.save_tsv . . . . .	144
7.238	TableList.save_xml . . . . .	144
7.239	TableList.set_changed . . . . .	145
7.240	TableList.set_format . . . . .	146
7.241	TableList.set_name . . . . .	146
7.242	TableList.set_readonly . . . . .	146
7.243	TableList.set_type . . . . .	147
7.244	TableList.structure . . . . .	147
7.245	TableList.view . . . . .	147
7.246	TimeDelta . . . . .	148
7.247	use_progress_indicators . . . . .	149
7.248	variance . . . . .	150
<b>8</b>	<b>Benfords</b>	<b>151</b>
8.1	analyze . . . . .	151
8.2	calc_benford . . . . .	153
8.3	get_expected . . . . .	154
<b>9</b>	<b>Crosstable</b>	<b>155</b>
9.1	pivot . . . . .	155
9.2	pivot_map . . . . .	158
9.3	pivot_map_detail . . . . .	159
9.4	pivot_table . . . . .	161

<b>10 Database</b>	<b>164</b>
10.1 MySQLConnection . . . . .	164
10.2 OdbcConnection . . . . .	165
10.3 OracleConnection . . . . .	166
10.4 PostgreSQLConnection . . . . .	166
10.5 PyGreSQLConnection . . . . .	167
10.6 SqliteConnection . . . . .	167
<b>11 Financial</b>	<b>169</b>
11.1 asset_turnover . . . . .	169
11.2 current_ratio . . . . .	169
11.3 debt_to_equity . . . . .	170
11.4 earnings_per_share . . . . .	170
11.5 inventory_turnover . . . . .	170
11.6 net_working_capital . . . . .	171
11.7 price_earnings . . . . .	171
11.8 profit_margin . . . . .	172
11.9 quick_ratio . . . . .	172
11.10return_on_assets . . . . .	172
11.11return_on_common_equity . . . . .	173
11.12return_on_equity . . . . .	173
<b>12 Grouping</b>	<b>174</b>
12.1 stratify . . . . .	174
12.2 stratify_by_date . . . . .	175
12.3 stratify_by_expression . . . . .	177
12.4 stratify_by_step . . . . .	177
12.5 stratify_by_value . . . . .	179
12.6 summarize . . . . .	179
12.7 summarize_by_date . . . . .	180
12.8 summarize_by_expression . . . . .	181
12.9 summarize_by_step . . . . .	182
12.10summarize_by_value . . . . .	183
<b>13 os</b>	<b>185</b>
13.1 execl . . . . .	185
13.2 execl . . . . .	186
13.3 execlp . . . . .	186

13.4	execlpe . . . . .	186
13.5	execvp . . . . .	187
13.6	execvpe . . . . .	187
13.7	getenv . . . . .	188
13.8	makedirs . . . . .	188
13.9	popen2 . . . . .	188
13.10	popen3 . . . . .	189
13.11	popen4 . . . . .	189
13.12	removedirs . . . . .	190
13.13	renames . . . . .	190
13.14	spawnl . . . . .	191
13.15	spawnle . . . . .	191
13.16	spawnlp . . . . .	192
13.17	spawnlpe . . . . .	192
13.18	spawnv . . . . .	193
13.19	spawnve . . . . .	193
13.20	spawnvp . . . . .	194
13.21	spawnvpe . . . . .	194
13.22	urandom . . . . .	195
13.23	walk . . . . .	195
<b>14</b>	<b>os.path</b>	<b>197</b>
14.1	abspath . . . . .	197
14.2	basename . . . . .	197
14.3	commonprefix . . . . .	197
14.4	dirname . . . . .	198
14.5	exists . . . . .	198
14.6	expanduser . . . . .	198
14.7	expandvars . . . . .	199
14.8	getatime . . . . .	199
14.9	getctime . . . . .	199
14.10	getmtime . . . . .	199
14.11	getsize . . . . .	200
14.12	isabs . . . . .	200
14.13	isdir . . . . .	200
14.14	isfile . . . . .	200
14.15	islink . . . . .	201
14.16	ismount . . . . .	201

14.17join . . . . .	201
14.18lexists . . . . .	201
14.19normcase . . . . .	202
14.20normpath . . . . .	202
14.21realpath . . . . .	202
14.22relpath . . . . .	202
14.23samefile . . . . .	203
14.24sameopenfile . . . . .	203
14.25samestat . . . . .	203
14.26split . . . . .	204
14.27splitdrive . . . . .	204
14.28splittext . . . . .	204
14.29walk . . . . .	205
<b>15 random</b>	<b>206</b>
15.1 Random . . . . .	206
15.2 Random.betavariate . . . . .	207
15.3 Random.choice . . . . .	207
15.4 Random.expovariate . . . . .	207
15.5 Random.gammavariate . . . . .	208
15.6 Random.gauss . . . . .	208
15.7 Random.getstate . . . . .	208
15.8 Random.lognormvariate . . . . .	209
15.9 Random.normalvariate . . . . .	209
15.10Random.paretovariate . . . . .	209
15.11Random.randint . . . . .	210
15.12Random.randrange . . . . .	210
15.13Random.sample . . . . .	211
15.14Random.seed . . . . .	211
15.15Random.setstate . . . . .	212
15.16Random.shuffle . . . . .	212
15.17Random.triangular . . . . .	212
15.18Random.uniform . . . . .	213
15.19Random.vonmisesvariate . . . . .	213
15.20Random.weibullvariate . . . . .	214
15.21SystemRandom . . . . .	214
15.22SystemRandom.betavariate . . . . .	214
15.23SystemRandom.choice . . . . .	215

15.24SystemRandom.expovariate . . . . .	215
15.25SystemRandom.gammavariate . . . . .	215
15.26SystemRandom.gauss . . . . .	216
15.27SystemRandom.getrandbits . . . . .	216
15.28SystemRandom.getstate . . . . .	216
15.29SystemRandom.jumpahead . . . . .	216
15.30SystemRandom.lognormvariate . . . . .	217
15.31SystemRandom.normalvariate . . . . .	217
15.32SystemRandom.paretovariate . . . . .	217
15.33SystemRandom.randint . . . . .	218
15.34SystemRandom.random . . . . .	218
15.35SystemRandom.randrange . . . . .	218
15.36SystemRandom.sample . . . . .	219
15.37SystemRandom.seed . . . . .	219
15.38SystemRandom.setstate . . . . .	220
15.39SystemRandom.shuffle . . . . .	220
15.40SystemRandom.triangular . . . . .	220
15.41SystemRandom.uniform . . . . .	221
15.42SystemRandom.vonmisesvariate . . . . .	221
15.43SystemRandom.weibullvariate . . . . .	222
15.44WichmannHill . . . . .	222
15.45WichmannHill.betavariate . . . . .	222
15.46WichmannHill.choice . . . . .	223
15.47WichmannHill.expovariate . . . . .	223
15.48WichmannHill.gammavariate . . . . .	223
15.49WichmannHill.gauss . . . . .	224
15.50WichmannHill.getstate . . . . .	224
15.51WichmannHill.jumpahead . . . . .	224
15.52WichmannHill.lognormvariate . . . . .	225
15.53WichmannHill.normalvariate . . . . .	225
15.54WichmannHill.paretovariate . . . . .	225
15.55WichmannHill.randint . . . . .	226
15.56WichmannHill.random . . . . .	226
15.57WichmannHill.randrange . . . . .	226
15.58WichmannHill.sample . . . . .	227
15.59WichmannHill.seed . . . . .	227
15.60WichmannHill.setstate . . . . .	228
15.61WichmannHill.shuffle . . . . .	228

15.62WichmannHill.triangular . . . . .	229
15.63WichmannHill.uniform . . . . .	229
15.64WichmannHill.vonmisesvariate . . . . .	229
15.65WichmannHill.weibullvariate . . . . .	230
15.66WichmannHill.whseed . . . . .	230
<b>16 re</b>	<b>231</b>
16.1 compile . . . . .	233
16.2 escape . . . . .	234
16.3 findall . . . . .	234
16.4 finditer . . . . .	234
16.5 match . . . . .	235
16.6 purge . . . . .	235
16.7 Scanner . . . . .	235
16.8 Scanner.scan . . . . .	236
16.9 search . . . . .	236
16.10split . . . . .	236
16.11sub . . . . .	237
16.12subn . . . . .	237
16.13template . . . . .	238
<b>17 Simple</b>	<b>239</b>
17.1 col_join . . . . .	239
17.2 col_left_join . . . . .	240
17.3 col_match . . . . .	241
17.4 col_match_diff . . . . .	241
17.5 col_match_same . . . . .	242
17.6 col_right_join . . . . .	243
17.7 compare_records . . . . .	244
17.8 custom_match . . . . .	245
17.9 custom_match_diff . . . . .	246
17.10custom_match_same . . . . .	247
17.11describe . . . . .	248
17.12expression_match . . . . .	249
17.13find_duplicates . . . . .	250
17.14find_gaps . . . . .	251
17.15fuzzycoljoin . . . . .	251
17.16fuzzymatch . . . . .	253

17.17fuzzysearch . . . . .	254
17.18get_unordered . . . . .	255
17.19join . . . . .	255
17.20left_join . . . . .	256
17.21regex_match . . . . .	257
17.22right_join . . . . .	258
17.23select . . . . .	259
17.24select_by_value . . . . .	260
17.25select_nonoutliers . . . . .	261
17.26select_nonoutliers.z . . . . .	262
17.27select_outliers . . . . .	263
17.28select_outliers.z . . . . .	264
17.29select_records . . . . .	264
17.30sort . . . . .	265
17.31soundex . . . . .	266
17.32soundexcol . . . . .	267
17.33transpose . . . . .	268
17.34wildcard_match . . . . .	268
<b>18 string</b>	<b>270</b>
18.1 atof . . . . .	270
18.2 atoi . . . . .	271
18.3 atol . . . . .	271
18.4 capitalize . . . . .	272
18.5 capwords . . . . .	272
18.6 center . . . . .	272
18.7 count . . . . .	273
18.8 expandtabs . . . . .	273
18.9 find . . . . .	273
18.10Formatter.check_unused_args . . . . .	274
18.11Formatter.convert_field . . . . .	274
18.12Formatter.format . . . . .	274
18.13Formatter.format_field . . . . .	275
18.14Formatter.get_field . . . . .	275
18.15Formatter.get_value . . . . .	275
18.16Formatter.parse . . . . .	276
18.17Formatter.vformat . . . . .	276
18.18index . . . . .	276



18.19join . . . . .	277
18.20joinfields . . . . .	277
18.21ljust . . . . .	277
18.22lower . . . . .	278
18.23lstrip . . . . .	278
18.24replace . . . . .	279
18.25rfind . . . . .	279
18.26rindex . . . . .	280
18.27rjust . . . . .	280
18.28rsplit . . . . .	280
18.29rstrip . . . . .	281
18.30split . . . . .	281
18.31splitfields . . . . .	282
18.32strip . . . . .	282
18.33swapcase . . . . .	283
18.34Template . . . . .	283
18.35Template.safe_substitute . . . . .	283
18.36Template.substitute . . . . .	283
18.37translate . . . . .	284
18.38upper . . . . .	284
18.39zfill . . . . .	284
<b>19 sys</b>	<b>286</b>
<b>20 Trending</b>	<b>288</b>
20.1 average_slope . . . . .	288
20.2 cusum . . . . .	289
20.3 handshake_slope . . . . .	290
20.4 highlow_slope . . . . .	290
20.5 regression . . . . .	291
<b>21 urllib</b>	<b>293</b>
21.1 addbase . . . . .	293
21.2 addbase.close . . . . .	294
21.3 addclosehook . . . . .	294
21.4 addclosehook.close . . . . .	294
21.5 addinfo . . . . .	294
21.6 addinfo.close . . . . .	295

21.7	addinfo.info	295
21.8	addinfourl	295
21.9	addinfourl.close	295
21.10	addinfourl.getcode	296
21.11	addinfourl.geturl	296
21.12	addinfourl.info	296
21.13	basejoin	296
21.14	ContentTooShortError	297
21.15	FancyURLopener	297
21.16	FancyURLopener.addheader	297
21.17	FancyURLopener.cleanup	297
21.18	FancyURLopener.close	298
21.19	FancyURLopener.get_user_passwd	298
21.20	FancyURLopener.http_error	298
21.21	FancyURLopener.http_error_301	299
21.22	FancyURLopener.http_error_302	299
21.23	FancyURLopener.http_error_303	300
21.24	FancyURLopener.http_error_307	301
21.25	FancyURLopener.http_error_401	301
21.26	FancyURLopener.http_error_407	302
21.27	FancyURLopener.http_error_default	303
21.28	FancyURLopener.open	303
21.29	FancyURLopener.open_data	303
21.30	FancyURLopener.open_file	304
21.31	FancyURLopener.open_ftp	304
21.32	FancyURLopener.open_http	304
21.33	FancyURLopener.open_https	305
21.34	FancyURLopener.open_local_file	305
21.35	FancyURLopener.open_unknown	305
21.36	FancyURLopener.open_unknown_proxy	306
21.37	FancyURLopener.prompt_user_passwd	306
21.38	FancyURLopener.redirect_internal	306
21.39	FancyURLopener.retrieve	307
21.40	FancyURLopener.retry_http_basic_auth	307
21.41	FancyURLopener.retry_https_basic_auth	308
21.42	FancyURLopener.retry_proxy_http_basic_auth	308
21.43	FancyURLopener.retry_proxy_https_basic_auth	309
21.44	ftperrors	309

21.45ftpwrapper . . . . .	309
21.46ftpwrapper.close . . . . .	310
21.47ftpwrapper.endtransfer . . . . .	310
21.48ftpwrapper.init . . . . .	310
21.49ftpwrapper.retrfile . . . . .	311
21.50getproxies . . . . .	311
21.51getproxies_environment . . . . .	311
21.52getproxies_macosx_sysconf . . . . .	311
21.53localhost . . . . .	312
21.54main . . . . .	312
21.55noheaders . . . . .	312
21.56pathname2url . . . . .	312
21.57proxy_bypass . . . . .	313
21.58proxy_bypass_environment . . . . .	313
21.59proxy_bypass_macosx_sysconf . . . . .	313
21.60quote . . . . .	314
21.61quote_plus . . . . .	314
21.62reporthook . . . . .	315
21.63splitattr . . . . .	315
21.64splithost . . . . .	315
21.65splitnport . . . . .	316
21.66splitpasswd . . . . .	316
21.67splitport . . . . .	316
21.68splitquery . . . . .	317
21.69splittag . . . . .	317
21.70splittype . . . . .	317
21.71splituser . . . . .	317
21.72splitvalue . . . . .	318
21.73test . . . . .	318
21.74test1 . . . . .	318
21.75thishost . . . . .	318
21.76toBytes . . . . .	319
21.77unquote . . . . .	319
21.78unquote_plus . . . . .	319
21.79unwrap . . . . .	319
21.80url2pathname . . . . .	320
21.81urlcleanup . . . . .	320
21.82urlencode . . . . .	320

21.83urlopen . . . . .	321
21.84URLopener . . . . .	321
21.85URLopener.addheader . . . . .	321
21.86URLopener.cleanup . . . . .	321
21.87URLopener.close . . . . .	322
21.88URLopener.http_error . . . . .	322
21.89URLopener.http_error_default . . . . .	322
21.90URLopener.open . . . . .	323
21.91URLopener.open_data . . . . .	323
21.92URLopener.open_file . . . . .	324
21.93URLopener.open_ftp . . . . .	324
21.94URLopener.open_http . . . . .	324
21.95URLopener.open_https . . . . .	324
21.96URLopener.open_local_file . . . . .	325
21.97URLopener.open_unknown . . . . .	325
21.98URLopener.open_unknown_proxy . . . . .	325
21.99URLopener.retrieve . . . . .	326
21.100urlretrieve . . . . .	326
<b>22 xml.etree.ElementTree</b>	<b>328</b>
22.1 Comment . . . . .	328
22.2 dump . . . . .	328
22.3 Element . . . . .	328
22.4 ElementTree . . . . .	329
22.5 ElementTree.find . . . . .	329
22.6 ElementTree.findall . . . . .	329
22.7 ElementTree.findtext . . . . .	330
22.8 ElementTree.getiterator . . . . .	330
22.9 ElementTree.getroot . . . . .	330
22.10ElementTree.parse . . . . .	330
22.11ElementTree.write . . . . .	331
22.12fixtag . . . . .	331
22.13fromstring . . . . .	331
22.14iselement . . . . .	332
22.15iterparse . . . . .	332
22.16iterparse.next . . . . .	332
22.17parse . . . . .	332
22.18PI . . . . .	333

22.19ProcessingInstruction . . . . .	333
22.20QName . . . . .	333
22.21SubElement . . . . .	334
22.22toString . . . . .	334
22.23TreeBuilder . . . . .	334
22.24TreeBuilder.close . . . . .	335
22.25TreeBuilder.data . . . . .	335
22.26TreeBuilder.end . . . . .	335
22.27TreeBuilder.start . . . . .	335
22.28XML . . . . .	336
22.29XMLID . . . . .	336
22.30XMLParser . . . . .	336
22.31XMLParser.close . . . . .	336
22.32XMLParser.doctype . . . . .	337
22.33XMLParser.feed . . . . .	337
22.34XMLTreeBuilder . . . . .	337
22.35XMLTreeBuilder.close . . . . .	338
22.36XMLTreeBuilder.doctype . . . . .	338
22.37XMLTreeBuilder.feed . . . . .	338
22.38Additional Libraries . . . . .	338
22.38.1 GUID . . . . .	339
22.38.2 Statistics . . . . .	340
22.38.3 Graphs and Plots . . . . .	340
22.38.4 Python Libraries . . . . .	340
<b>A Creating Detectlets</b>	<b>343</b>
A.1 The Detectlet Process . . . . .	344
A.2 Detectlet Anatomy . . . . .	345
A.3 Detectlet Version . . . . .	345
A.4 The wizard variable . . . . .	346
A.5 Analysis Function . . . . .	348
A.6 Example Input . . . . .	349
<b>B Creating Plugins</b>	<b>351</b>
<b>C Example Picalo Scripts</b>	<b>353</b>
C.1 Discovery of Outliers . . . . .	353
C.2 Identifying Phantom Vendors . . . . .	354

C.3 Finding Unapproved Vendors . . . . .	356
C.4 Stratification and Summarization . . . . .	357
C.5 Standardizing the Time Axis . . . . .	359

# About This Manual

The purpose of this manual is to describe the more advanced interfaces of Picalo. For example, the shell and the scripting areas of Picalo require you to know some scripting, but these areas also offer more power than the regular menus. This manual is for users who want more power out of Picalo or who want to write detectlets.

## A Moving Target

Since Picalo is a work in progress, it is constantly improving and changing. Therefore, the dialogs in the program may be slightly different that what you see in this manual due to change in the program. We'll try to keep the manual updated with the program, but we simply do not have enough time to update the manual in perfect unison with the program. Since the Picalo program itself is the main priority, the manual may be slightly behind the actual features or user interface of the program. Please check the current API documentation for Picalo for updates.

Update August 2007: This manual is about a year old now. Most of what it says is still correct, but there may be things that need updating. The most updated manual about Picalo is the Picalo Cookbook – you may want to read it first.

# Chapter 1

## Introduction

The primary introduction to Picalo is contained in the Introductory Manual. Please read through the Introductory Manual before going into this manual.

### 1.1 Picalo Architecture

Picalo is built upon an open architecture (Figure 1.1), although not all parts are necessarily open source. The following diagram describes the different parts of the Picalo platform.

#### Levels:

- Level 1 Routines (production): This open source level includes all the basic data structures in Picalo. It was finished in the 2001-2002 time frame, and while we continue to add some routines to it, is quite stable and finished.
- Level 2 Routines (development): This level allows non-technical people to run advanced analyses without having to script or write programs. Picalo's pluggable architecture allows new Detectlets to be installed in the existing program. We hope that Detectlet libraries will be made available by individuals and/or companies – the license and price is up to the developers.
- Level 3 Routines (planned): When developed, this level will apply expert system rules to intelligently run level 2 routines (Detectlets) to



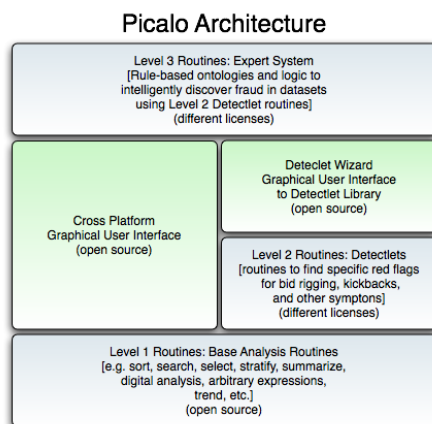


Figure 1.1: Picalo Architecture

discover many different types of fraud on its own. The license for the expert system is not decided at this point.

## Chapter 2

# Picalo Tables

Tables are the primary data structure of Picalo. You'll load your data into Tables for analysis. Tables allow you to load data from different sources (delimited text files, databases, etc.), manipulate data types, run analyses, group data, and save results<sup>1</sup>.

This section describes how to work with tables from the Shell or from a script. Most of the commands shown here can also be done from the Picalo File menu.

One of the primary features of tables is their headings. Each column in a table *always* has a unique name as well as an index. It is not possible to have a table without column headings. If data are loaded from database queries, the column headings are automatically gleaned from table schema and/or SQL query statements. If data are loaded from delimited text files, the first row in a table can specify column names to be used as headings. If no headers are found, Picalo creates default headings like col000, col001, and so forth.

Table rows (termed “records”) are accessed via their index. Records are not named. Table cells are accessed first via row, then via column. The following are examples of good and bad table access:

```
1 # cell value from first row, first column
2 table[0][0]
3 # cell value from the tenth row, third column
4 table[9][2]
```

---

<sup>1</sup>For advanced readers familiar with Python, a table imitates a list of lists. In fact, the Table, Record, and Column classes all extend list. Most routines (although not all) allow you to use tables and list of lists interchangeably. However, tables are much richer than lists: tables contain methods to access columns, calculate new columns, and do many other things not available to Python lists.

```

5 # cell value from fifth row, column with name: 'address'
6 table[4]['address']
7 # cell value from fifth row, column with name: 'address'
8 table['address'][4]

```

**Zero-Based Indices** As seen in the above example, record and column indices in Picalo are zero-based. The first record is `table[0]` and the first column is `record[0]`. Zero-based indices take some getting used to. They are used in Picalo to be consistent with the large majority of scripting languages in the world.

## 2.1 Creating Tables

Tables can be created in many different ways. This section describes the three most common ways: new tables, loading tables from file, and retrieving from databases.

**Creating New, Empty Tables** Tables can be created by specifying either then number of columns or the column heading names. If you do not specify the column heading names, Picalo will assign default heading names in the format: `col000`, `col001`, `col002`, etc. Table creation methods are shown in the following example:

```

1 # creates a new table with 5 columns (named col000 through col004) of string type
2 table1 = Table(5)
3 # creates a new table with 3 columns by specifying the
4 # column heading names and types
5 table2 = Table([
6     ( 'ID', int ),
7     ( 'Name', unicode ),
8     ( 'Salary', float ),
9 ])

```

While you always have to specify column headings, it is optional to specify the initial data to be placed into a table. Tables are initialized with data from a list of lists or from another table. This can be done in the following ways:

```

1 # creates the table with initial data
2 listoflists = [
3     [ 1, 'Dan', 50000 ],
4     [ 2, 'Ben', 15000 ],
5     [ 3, 'Alice', 500000 ]
6 ]
7 table1 = Table([
8     ( 'ID', int ),
9     ( 'Name', unicode ),
10    ( 'Salary', float ),

```

```

11 ], listoflists)
12 # creates a new table, initialized with data fom table1
13 table2 = Table(3, table1)
14 # creates a new table, then appends the data with extend()
15 table3 = Table([
16     ( 'ID', int ),
17     ( 'Name', unicode ),
18     ( 'Salary', float ),
19 ])
20 table3.extend(table1)

```

**Loading and Saving Text Files** Most data comes in text files. If you are working with an IT department for data access, they'll likely send requested data in delimited text files, fixed-width files, or XML files on a CD/DVD or over the Internet. Most proprietary applications, such as Excel, can export (Save As...) data to these formats.

The Dataset module includes three routines to import delimited text files into tables. While Picalo handles all the details of these files, it is useful to understand how they are formatted. Delimited text files contain one record per line, with fields separated by a *delimiter*. In CSV files, this delimiter is a comma. In TSV files, this delimiter is the tab character. You can also specify a different delimiter if you have a specialty file that doesn't match the CSV or TSV standards. Delimited files also define a *qualifier*. The qualifier character (usually a single quote or double quote) is only important when a field contains the delimiter character. For example, suppose a field in a CSV file contains "Hello, world". The presence of the comma in the actual value of the field will confuse the import and export parser. Therefore, the field is wrapped in a qualifier so the parser can determine when a comma is a delimiter and when a comma is part of the actual field text.

The following shows an example of a CSV file with a header row (exported from MS Excel):

```

1 ID,Name,Salary,Quote
2 1,Dan,50000,"Bring it on!"
3 2,Ben,15000,Love you baby
4 3,Alice,500000,Unknown

```

Notice that fields are separated by commas, and Dan's quote is surrounded by qualifiers (since it contains the delimiter character).

The `load_delimited`, `load_csv`, and `load_tsv` routines take filename and a header row flag and produce a new table. The following shows an example of loading tables from delimited text files:

```

1 # note the use of forward slashes on all operating systems, including Windows
2 table1 = load_csv('c:/temp/mydata.csv')

```

Please consider the following when loading data from delimited text files:

1. Delimited text files do not contain any data type information. This is not a limitation of Picalo, but rather a limitation of the fact that delimited text files simply do not contain type information. Therefore, Picalo types all fields as strings. You need to perform any needed conversions after loading the table, such as strings to dates, strings to numbers, etc. See the `table.set_type` method for this conversion.
2. The Dataset module loads the entire table into memory. It is limited by your computer's real and virtual memory. If you use the Database module (discussed later) to iterate through records instead of direct tables, data are only loaded when used. The Database module can thus handle vast amounts of data without overloading your memory. However, tables provide much richer functionality than using databases directly, so you should always work with tables when possible. For example, tables can be created directly from database result sets (see the Database section in the Reference chapter). In practical terms, Picalo has shown it can load millions of records into a table at once without problems.

Picalo can also import Excel .xls files. Use `load_excel` to load Microsoft Excel files.

Once tables are created, they can be saved to delimited text files with the `save_delimited`, `save_csv`, and `save_tsv` methods. These are called directly on the table object, as seen in the following example:

```
1 # load the table from a Unix/Mac drive
2 table1 = load_csv('/tmp/simpsons.csv')
3 # save the table as TSV to a Windows drive
4 table1.save_tsv('c:/temp/simpsons.tsv')
```

**Retrieving Tables from Databases** Tables can be created by running queries against databases with the Database module. See the database chapter for more information on retrieving tables from databases.

## 2.2 Columns and Headings

As described in Chapter 2, Picalo tables *always* contain heading names and types. Tables support many different column and heading operations. Col-

umn headings are objects that contain the column name, a type, and (optional) calculation.

Column names are case sensitive. This means that `rec['ID']` is different than `rec['id']` or `rec['Id']`.

Column information, such as heading objects, names, and count can be retrieved with the command shown in the following example:

```

1 # create a simple table
2 table1 = Table([
3     ( 'ID', int ),
4     ( 'Name', unicode ),
5     ( 'Salary', float ),
6 ], [
7     [ 1, 'Dan', 50000],
8     [ 2, 'Ben', 15000],
9     [ 3, 'Alice', 500000]
10 ])
11
12 # retrieve the column heading list and print name
13 headings = table1.get_columns()
14 for heading in headings: # heading variable will be each column object
15     print heading.name
16 # prints:
17 #     ID
18 #     Name
19 #     Salary
20
21 # retrieve only the column names
22 names = table1.get_column_names()
23 # names is now a list: ['ID', 'Name', 'Salary']
24
25 # get the number of columns in the table
26 width = table1.column_count()
27 # width is now 3

```

**Adding and Removing Columns** You can append columns to the end of the table (i.e. to the right side) or insert columns at any place in the table. You can also delete entire columns. When columns are added or removed, the indices of remaining columns change to reflect the new sequence. Column names remain unchanged and always stay with their respective columns.

The following example shows these routines:

```

1 # create a simple table
2 table1 = Table([
3     ( 'ID', str ),
4     ( 'Name', unicode ),
5     ( 'Salary', float ),
6 ], [
7     [ 1, 'Dan', 50000],
8     [ 2, 'Ben', 15000],
9 ])
10

```

```

11 # add a new 'quote' column of type string
12 table1.append_column('Quote', str)
13 # table1 is now:
14 # +-----+-----+-----+
15 # | ID | Name | Salary | Quote |
16 # +-----+-----+-----+
17 # | 1 | Dan | 50000 | None |
18 # | 2 | Ben | 15000 | None |
19 # +-----+-----+-----+
20
21 # delete the Salary column
22 table1.delete_column('Salary')
23 # table1 is now:
24 # +-----+-----+
25 # | ID | Name | Quote |
26 # +-----+-----+
27 # | 1 | Dan | None |
28 # | 2 | Ben | None |
29 # +-----+-----+
30
31 # insert a new column in position 1
32 # this one includes values for the new column
33 table1.insert_column(1, 'IQ', int, [100, 75])
34 # table1 is now:
35 # +-----+-----+-----+
36 # | ID | IQ | Name | Quote |
37 # +-----+-----+-----+
38 # | 1 | 100 | Dan | None |
39 # | 2 | 75 | Ben | None |
40 # +-----+-----+-----+

```

**Calculated Columns** In addition to regular columns, new columns can be calculated with a function. Their value is always determined by the value of other fields in each record. Active calculated fields are always updated – their value is dynamically calculated each time they are accessed. Therefore, calculated fields always reflect any changes made to regular columns in a table.

A creative use of calculated columns is *type conversion* (note that Picalo tables also include a `set_type` method). Rather than modifying the actual values in a column, simply create a new column with an expression that converts the type in real time. This method of type conversion is not particularly efficient, but it is easy. For example, if you’ve read a TSV table (which reads all columns as strings) that includes an Amount column, create an AmountN column that contains the numeric equivalent of the strings in the Amount column:

```

1 table.append_calculated('AmountN', float, "float(Amount)")

```

Actual column values for calculated columns are ignored. Instead, they are calculated each time they are accessed by running the expression associ-

ated with their column. Therefore, Picalo won't say anything if you set the value of individual cells in calculated columns, but it will not return them or use them. In other words, the actual values of calculated columns are irrelevant; the associated function is the important factor.

Calculated functions can use the column names of the table to represent the other fields in the given row. Following is an example:

```

1 # create a simple table
2 table1 = Table([
3   ( 'ID', int ),
4   ( 'Name', unicode ),
5   ( 'Salary', float ),
6 ], [
7   [ 1, 'Dan', 50000],
8   [ 2, 'Ben', 15000],
9 ])
10
11 # calculate twice salary
12 table1.append_calculated('DoubleSalary', int, "Salary*2")
13 # table1 is now:
14 # +-----+-----+-----+-----+
15 # | ID | Name | Salary | DoubleSalary |
16 # +-----+-----+-----+-----+
17 # | 1 | Dan | 50000 | 100000 |
18 # | 2 | Ben | 15000 | 30000 |
19 # +-----+-----+-----+-----+
20 table1.delete_column('DoubleSalary')
21
22 # calculate a random number column
23 # using Python's random library
24 import random
25 table1.insert_calculated(1, 'Random', float, "random.randint(0, 10)")
26 # table1 is now:
27 # +-----+-----+-----+-----+
28 # | ID | Random | Name | Salary |
29 # +-----+-----+-----+-----+
30 # | 1 | 3 | Dan | 50000 |
31 # | 2 | 6 | Ben | 15000 |
32 # +-----+-----+-----+-----+

```

Note that the random number will be regenerated each time you view the table. For a more permanent random number, see the next paragraph.

**Static Calculated Columns** In the last example (using the random library) the random number will constantly change in the Random column. In other words, each time you access the value, the random formula will be evaluated and a new random number will be returned.

If you want to calculate the random number only once, use a static calculated column. A static calculated column will record the *results* of the given formula rather than the formula itself. The calculation will be run only once



and the values will be static after that.

For most analyses, you should use static calculated columns. They are faster because they don't have to update. If your data isn't changing anyway, static calculated columns are the way to go. Only use active calculated columns when you really need the formulas to update (which means your other data are changing).

Following is the more persistent random example:

```

1 # calculate a static random number column
2 # using Python's random library
3 import random
4 table1.insert_calculated_static(1, 'Random', int, "random.randint(0, 10)")
5 # table1 is now:
6 # +-----+-----+-----+
7 # | ID | Random | Name | Salary |
8 # +-----+-----+-----+
9 # | 1 | 3 | Dan | 50000 |
10 # | 2 | 6 | Ben | 15000 |
11 # +-----+-----+-----+

```

Note that static random columns are in all ways equivalent to regular columns after their creation. The formula is not saved after creation.

**Accessing Individual Columns** Picalo tables provide a number of ways to access individual columns. Columns become a list of values just like any other Python list. This is useful in statistical or other routines that require a list of numbers (most often found in a table column). The data in the column are live: changes to the underlying table are reflected in the column object, and changes to the column object are reflected in the underlying table.

The following example shows the use of a column:

```

1 # create a simple table
2 table1 = Table([
3     ( 'ID', int ),
4     ( 'Name', unicode ),
5     ( 'Salary', float ),
6 ], [
7     [ 1, 'Dan', 50000 ],
8     [ 2, 'Ben', 15000 ],
9     [ 3, 'Alice', 500000 ]
10 ])
11
12 # retrieve the name column as a list
13 col = table1['Name']
14 # col is now: [ 'Dan', 'Ben', 'Alice' ]
15 col[1] = 'Sally' # second row
16 # table1 is now:
17 # +-----+-----+-----+
18 # | ID | Name | Salary |
19 # +-----+-----+-----+

```

20	#		1		Dan		50000	
21	#		2		Sally		15000	
22	#		3		Alice		500000	
23	#	+	-----	+	-----	+	-----	+

## 2.3 Records

As you might expect, a Picalo record is a horizontal row in a table. It is important for readers to understand the difference between a table in spreadsheet format and a table in database format. Spreadsheet tables are quite free-form. Cells are calculations of other cells, and data is often two or even three dimensional.

Picalo tables are normally retrieved from databases or from text (CSV) files. They share the same assumptions as relations/tables in relational database theory. Database relations are more structured than spreadsheet tables. A row of the relation (called a record in Picalo) always represents an entity: a transaction, a person, a sale, a timecard clock-in, etc. normally keep only raw data – cells are not usually calculated in the source data.

Each record in a database relation contains a key: a unique column value (or set of column values) that uniquely identifies the record from other records. Key column(s) are useful when grouping and summarizing records according to person, invoice, or sale.

Records can be retrieved from a table using bracket notation—the same way list items are accessed in standard Python. The records are zero-based. Records are active: modifications to the record fields are reflected in the table, and changes in the table are reflected in the record. This is shown in the following example:

```

1 # create a simple table
2 table1 = Table([
3   ( 'ID', int ),
4   ( 'Name', unicode ),
5   ( 'Salary', float ),
6 ], [
7   [ 1, 'Dan', 50000],
8   [ 2, 'Ben', 15000],
9   [ 3, 'Alice', 500000]
10 ])
11
12 # retrieve the first record
13 rec = table1[0]
14 # rec is now [ 1, 'Dan', 50000 ]
15
16 # change the values of fields in rec

```

```

17 rec['ID'] = 5
18 rec[1] = 'Sally'
19
20 # change a value directly from the table
21 table1[0]['Salary'] = 100000
22
23 # table1 is now:
24 # +-----+-----+
25 # | ID | Name | Salary |
26 # +-----+-----+
27 # | 5 | Sally | 100000 |
28 # | 2 | Ben | 15000 |
29 # | 3 | Alice | 500000 |
30 # +-----+-----+
31
32 # use Python's slice notation (see Python tutorial for slices)
33 sub = rec[0:2]
34 # sub is now [ 5, 'Sally' ]
35 rec[1:3] = ['Maggie', 0]
36
37 # table1 is now:
38 # +-----+-----+
39 # | ID | Name | Salary |
40 # +-----+-----+
41 # | 5 | Maggie | 0 |
42 # | 2 | Ben | 15000 |
43 # | 3 | Alice | 500000 |
44 # +-----+-----+

```

New records can be added to tables in two ways: by appending/inserting new records or by appending entire tables (called extending). Appending a record inserts the new record to the end of the table. Inserting a record inserts the record at the index you specify. Otherwise, these two methods are identical. The **append** and **insert** methods allow several formats for cell value specification. Both methods return a reference to the new record, which you can ignore if you don't need it. These different methods of adding records are shown in the following example:

```

1 # create a simple table
2 table = Table([
3     ('ID', int),
4     ('Name', unicode),
5     ('Salary', float),
6 ], [
7     [ 1, 'Dan', 50000],
8     [ 2, 'Ben', 15000]
9 ])
10
11 # append records
12 table.append(3, 'Alice', 500000) # reg. parameters
13 table.append([4, 'Alice', 500000]) # a list
14 table.append(ID=5, Salary=500000, Name='Alice') # named cols
15 table.append({'ID':6, 'Name':'Alice', 'Salary':500000}) # a dictionary
16 table.append({0:7, 2:500000, 1:'Alice'}) # a dictionary with indices
17
18 # table is now:

```

```

19 # +-----+-----+
20 # | ID | Name | Salary |
21 # +-----+-----+
22 # | 1 | Dan | 50000 |
23 # | 2 | Ben | 15000 |
24 # | 3 | Alice | 500000 |
25 # | 4 | Alice | 500000 |
26 # | 5 | Alice | 500000 |
27 # | 6 | Alice | 500000 |
28 # | 7 | Alice | 500000 |
29 # +-----+-----+
30
31 # recreate the table
32 table = Table([
33     ( 'ID', int ),
34     ( 'Name', unicode ),
35     ( 'Salary', float ),
36 ], [
37     [ 1, 'Dan', 50000],
38     [ 2, 'Ben', 15000],
39 ])
40
41 # insert records (only difference from append
42 # is the index location—specified as first parameter)
43 table.insert(1, 3, 'Alice', 500000) # reg. parameters
44 table.insert(1, [4, 'Alice', 500000]) # a list
45 table.insert(1, ID=5, Salary=500000, Name='Alice') # named cols
46 table.insert(1, {'ID':6, 'Name':'Alice', 'Salary':500000}) # a dictionary
47 table.insert(1, {0:7, 2:500000, 1:'Alice'}) # a dictionary with indices
48
49 # table is now:
50 # +-----+-----+
51 # | ID | Name | Salary |
52 # +-----+-----+
53 # | 1 | Dan | 50000 |
54 # | 7 | Alice | 500000 |
55 # | 6 | Alice | 500000 |
56 # | 5 | Alice | 500000 |
57 # | 4 | Alice | 500000 |
58 # | 3 | Alice | 500000 |
59 # | 2 | Ben | 15000 |
60 # +-----+-----+

```

Entire tables can be added together with the `extend` method. Picalo will add tables of any type together, even if the table structures are different (such as different data types or numbers of columns). You need to ensure that tables you add together have the same number of columns, header names and types, and so forth. This is shown in the following example:

```

1 # create the first table
2 table1 = Table([
3     ( 'ID', int ),
4     ( 'Name', unicode ),
5     ( 'Salary', float ),
6 ], [
7     [ 1, 'Dan', 50000],

```

```

8  [ 2, 'Ben', 15000],
9  ])
10 # create the second table
11 table2 = Table([
12   ( 'ID', int ),
13   ( 'Name', unicode ),
14   ( 'Salary', float ),
15 ], [
16   [ 3, 'Alice', 500000]
17 ])
18
19 # append the records of table 1 to table 2
20 table2.extend(table1)
21
22 # table1 is unchanged
23 # table2 is now:
24 # +-----+
25 # | ID | Name | Salary |
26 # +-----+
27 # | 3 | Alice | 500000 |
28 # | 1 | Dan  | 50000  |
29 # | 2 | Ben  | 15000  |
30 # +-----+

```

Table can also be added and subtracted with the `+` and `-` operators<sup>2</sup>. When these operators are used, a new table is created and the two original tables are not modified. The subtract operator returns all records in the first table that are *not* in the second table. This is best shown with an example:

```

1 # create the first table
2 table1 = Table([
3   ( 'ID', int ),
4   ( 'Name', unicode ),
5   ( 'Salary', float ),
6 ], [
7   [ 1, 'Dan', 50000],
8   [ 2, 'Ben', 15000]
9 ])
10 # create the second table
11 table2 = Table([
12   ( 'ID', int ),
13   ( 'Name', unicode ),
14   ( 'Salary', float ),
15 ], [
16   [ 3, 'Alice', 500000]
17 ])
18
19 # create a new table
20 table3 = table1 + table2
21 # table1 and table2 are unchanged
22 # table3 is now:
23 # +-----+
24 # | ID | Name | Salary |

```

---

<sup>2</sup>For those familiar with C or Python programming, the `+=` and `-=` operators are also supported

```

25 # +-----+-----+-----+
26 # | 1 | Dan | 50000 |
27 # | 2 | Ben | 15000 |
28 # | 3 | Alice | 500000 |
29 # +-----+-----+-----+
30
31 # subtract table2 from table1:
32 table4 = table3 - table1
33 # table4 is now:
34 # +-----+-----+-----+
35 # | ID | Name | Salary |
36 # +-----+-----+-----+
37 # | 3 | Alice | 500000 |
38 # +-----+-----+-----+

```

Because database records represent individual entities, most analyses walk through the records from top to bottom, one by one. Each record is analyzed for the value of its fields. This is called *iteration*. See the Python Tutorial for more information on iteration through lists (Picalo tables iterate just like regular Python lists).

```

1 # create a simple table
2 table1 = Table([
3     ( 'ID', int )
4 ], [
5     [ 1 ],
6     [ 2 ]
7 ])
8
9 # iterate through the records
10 for rec in table1:
11     print rec.ID
12 # prints:
13 # 1
14 # 2
15
16 # iterate using the index and a range
17 # this is an alternative way to iterate
18 for i in range(len(table1)):
19     print table1[i].ID
20 # prints:
21 # 1
22 # 2
23
24 # finally, iterate through the index using Python's enumerate
25 for i, rec in enumerate(table1):
26     print i, rec.ID
27 # prints:
28 # 0 1
29 # 1 2

```

The Simple module contains additional operations on tables, such as selecting records based on a function, sorting, etc. See this module in the reference section for more information on these routines.

## 2.4 Viewing Tables

The primary way to view table data is via the `table.view` method. This method opens a graphical view of the table in a nice spreadsheet within the Picalo interface. In addition, *the data is editable* directly from within the spreadsheet. You can add columns, rows, and modify data by editing a cell's value. The spreadsheet contains a menu that provides a nice interface to your tables.

Important: Tables remain in memory whether you are viewing them or not. To actually remove a table from memory, use the `del tablename` command or right click the table and remove it from your project. In addition, Picalo does not automatically save tables you create. If you shut the program down without saving your tables, you'll lose your data. Since thousands of tables can be created by different Picalo functions, it is not practical or desirable to have Picalo automatically save to disk. Be sure to periodically back up your work.

# Chapter 3

## Data Types

Data Types refers to the type of data a variable, cell, or column can hold. Most databases have a set type for a given column, such as `varchar` to hold text, `integer` to hold numbers, and so forth. Number values can be added together while text values can be searched. It is important to send the correct type of data into each function.

Because of the importance of data types, the reference chapter later in this manual shows the data type of every function parameter. For example, strings (a.k.a. text) are denoted with `<str>` and Picalo tables are denoted with `<Table>`.

If you load data from a database connection (e.g. through a `SELECT` call in SQL), the data will retain the type they had in the database. For example, if you select a `VARCHAR(20)` field, the data will be strings; if you select an `INTEGER` field, the data will be ints. Keeping your data in a database is optimal because it always retains type information.

In like manner, if you load and save your data to and from native Picalo files, data will retain their types. Native Picalo files keep all changes you make to a table, including calculated columns, type conversion functions, and so forth.

The remainder of this chapter describes some of the basic types in Picalo.

### 3.1 Table

Picalo's table type is the core entity of the program. Almost all functions take tables as parameters and return tables from their routines. Chapter 2



of this manual describes tables in detail.

## 3.2 String

String variables hold any type of text, including characters, words, numbers, and paragraphs. Strings are the most common type of data in Picalo, but they are probably the least powerful. Strings support a limited set of functions (such as fuzzy matching). There is no inherent limit to the size of a string in Picalo.

Strings are denoted as **str** or **unicode**. The first version is for older, 8-bit strings and the latter is the international version.

```
1 t = Table(2, [['Dan', '2'], ['Stan', '4']])
2 t.set_type('col001', unicode)
3 name = str(1234)
```

## 3.3 Number Types

Picalo includes a set of number types to represent the many different types of numbers in the world. The two primary types of numbers used in databases are integers and floating point numbers.

Integers are numbers without the decimal point. For example, 1, 5, -6, and 15,200 are all integer numbers. Integer numbers are excellent to work with because they are exact, fast, and obvious. Picalo includes two integer types: **int** and **long**. The **long** type can hold larger numbers, but it takes more memory. In practice, Picalo will convert **int** numbers to **long** automatically when needed, so you should just work with **int**.

Floating point numbers include a decimal or fraction part. For example, 3.14, 5.50, and 27.2341 are all floating point numbers. Floating point numbers are not perfectly exact (some rounding and approximation occurs in the decimal portion). Picalo includes the **float** type to convert numbers to floating point numbers. It also includes the **money** type which formats numbers ‘prettier’ than raw floats. Otherwise the two types are identical.

The following examples show different types of numbers in Picalo:

```
1 i = 1    # i is an int with value 1
2 i = 1.0  # i is a float with value 1
3 i = int('1') # converts the string '1' to the int 1
4 f = float(i) # converts i to a floating point number
5 i = int('a') # causes an error since a is not a number
```

### 3.4 DateTime

Most variables are straightforward: strings, numbers, etc. Dates are another story. Ease of use has never been an advertised quality of any culture’s date system. The widely-accepted Gregorian calendar system is one of the most difficult systems available! Granted, adjustments are required each year due to the rotation of the Earth relative to its rotation around the Sun, but the addition of different number of days per month, leap years, leap centuries, and other factors make date math and analysis difficult. To make matters worse, the difference in date formats around the world further complicates the situation (is month expressed before or after the day? is the month a number or three letters or a full word?). Perhaps someday we’ll be on Internet time (ticks) or another easier system, but for now facts are what they are.

When data are loaded from delimited text files (csv, tsv, etc.), all columns are typed as strings. This means that “25 Jan 2004” + 1 = “25 Jan 20041”, similar to the way “abc” + “def” = “abcdef”. You need to convert date columns to actual date objects (DateTime) or real numbers so they can be used in analyses.

The optimal way to handle dates is to call `set_type` on the date column and let Picalo autodiscover the date format. Picalo can handle about 20 date formats for date alone, plus the same 20 formats with time added. See the DateTime object in the reference section for more information about the date formats.

The following example shows the conversion of the date column in a set of data:

```
1 >>> data = Table([( 'Name', str), ( 'Birthdate', str)], [
2 ... [ 'Danny', 'January 5, 1962'],
3 ... [ 'Betty', '6/11/1973'],
4 ... [ 'Joe', '2005-12-31'],
5 ... ])
6 >>> data.view()
7 +-----+
8 | Name | Birthdate |
9 +-----+
10 | Danny | January 5, 1962 |
11 | Betty | 6/11/1793 |
12 | Joe | 2005-12-31 5:30:14 pm |
13 +-----+
14 >>> data.set_type("Birthdate", DateTime)
15 >>> data.view()
16 +-----+
17 | Name | Birthdate |
18 +-----+
19 | Danny | 1962-01-05 00:00:00.00 |
20 | Betty | 1973-06-11 00:00:00.00 |
```

```

21 | Joe      | 2005-12-31 17:30:14.00 |
22 +-----+

```

Notice in the above example how the Birthdate column starts out as strings, similar to the way dates are loaded from text files. The call to `set.type` creates consistent `DateTime` objects. These new values can be added, compared, and analyzed effectively. The new values have a component for date and for time, although the time value is set to midnight for the cells that didn't include time.

Also note that this example has three different date formats to show the variety of input formats the `DateTime` can convert from. In real data, date columns usually use a consistent format from row to row.

### 3.5 None

Picalo defines the 'None' type to delineate when a cell has no value. This is similar to the use of NaN or Undefined values in other applications. You can use the `None` keyword in most places in Picalo scripts:

```

1 a = None
2 l = [1, 2, None, 3, None, 4]

```

Many scripts use the keyword `None`. For example, the `Crosstable.pivot()` method usually encounters many cells in the pivoted table that have no entries. These cells are set to `None`.

Note that `None` is not zero and `None` is not the empty string (''). Zero indicates a number value of zero on the number scale. The empty string denotes a string type with no text (such as a person without a middle name). The `None` type has a specific meaning. It means that no data exists for a given value.

### 3.6 Error

The error type is given to a cell when an error occurs in a calculation. Errors always show in tables as `<err>`. You can inspect the error type to know what caused the error with the `get_message()` method, as shown in the following example:

```

1 >>> data = Table(1, [[1],[2],[ 'a' ],[4]])
2 >>> data.view()
3 +-----+
4 | col000 |

```

```

5  +-----+
6  |         1 |
7  |         2 |
8  |    a      |
9  |         4 |
10 +-----+
11 >>> data.set_type('col000', float)
12 >>> data.view()
13 +-----+
14 | col000 |
15 +-----+
16 |    1.0  |
17 |    2.0  |
18 |  <err>  |
19 |    4.0  |
20 +-----+
21 >>> data[2].col000.get_message()
22 'invalid literal for float(): a'

```

Always check your results for the presence of `<err>` in cells. It is the primary way Picalo reports problems to you.

## 3.7 Other Types

Picalo (through Python) contains many more types than can be included in this manual. For example, Tables can hold file objects, open sockets, lists, dictionaries, and even other Picalo Tables! You can even build new types by creating classes in Picalo.

All of these concepts are for advanced Picalo programmers. See the Python tutorial and documentation for more information on using and creating extended types.

# Chapter 4

## Databases

Picalo provides access to relational databases through the Database module. It comes with the drivers for three types of connections: ODBC, PostgreSQL, and MySQL. The ODBC driver can connect to any type of database you can set up an ODBC connection for.

There are two primary ways you can get data: query it yourself or work through IT personnel. While you might be relegated to IT personnel because of security restrictions, it is significantly more efficient to query data yourself. Analysis work is normally iterative in nature: query data, analyze, requery, analyze, requery, ... The significant amount of noise in most data sets requires this requery process to weed out unimportant but nonstandard records that come up in the first iterations. If you have to go back to IT support on each iteration, you'll spend much of your time waiting for data. Direct access (through ODBC or Python drivers) to databases allows you to short circuit this process and spend your time in analysis.

Most importantly, iteration through data retrieved directly from databases does not require that all records be in memory at the same time. You can analyze terabytes without using much memory on your local computer (assuming your database server can handle it). The drawback of loading only required records into memory is you are limited to simple iteration: one record at a time in sequential order.

### 4.1 Creating the Connection

A Picalo database connection is created by calling the Database module. The following example shows a connection to the *PostgreSQL* database(shameless

plug: PostgreSQL is an excellent, free, ACID-compliant database that works well for data warehouses and fraud-detection databases):

```
1 # import picalo and psycopg db driver
2 from picalo import *
3
4 # first create the connection
5 conn = Database.PostgreSQLConnection(database='fraudratios')
6
7 # use the connection here
8
9 # close the connection
10 conn.close()
```

## 4.2 Querying Data

Relational databases use SQL (Structured Query Language) to query (e.g. select) records from a database. Many books on SQL exist, and I suggest you spend a day or two learning how to work with databases and SQL. The base concepts of the language are not hard, and querying of data is such a common task that it will be time well spent.

Python databases use the concept of a cursor (cursors are used in most other data-aware languages as well). A cursor is a pointer to a database that runs over a connection. Picalo allows you to ignore cursors altogether if you want. Although ignoring cursors uses slightly more resources on the database than a well-programmed script would, using the connection directly is much easier. This manual uses the connection directly, although all of the methods described can also be used on a cursor. In fact, Picalo database connections and cursors contain all the methods of regular Python connections and cursors.

Picalo allows the selection of data through two methods: **query** and **table**. The table method selects records into a Picalo table as described in Chapter 2. Once you have a table, you can peruse the data in any direction, change data (locally, not in the database itself), add columns, calculate columns, etc. Some Picalo routines also require real Picalo tables. If you are working with relatively small record sets (up to millions of records—is that exact enough? :), you should use tables as they are rich and full-featured.

If you are working with significant amounts of data, or if you simply need to iterate through query results sequentially (top to bottom), use the **query** method for a more efficient operation. The query method returns only an iterator to records. Record cells can be accessed by column name or index, but no other operations are permitted. As stated earlier, the **query** command

can handle terabytes of data, as long as your database and driver can handle it.

```

1 from picalo import *
2
3 # connect to the database
4 # first create the connection
5 conn = Database.PostgreSQLConnection(database='fraudratios')
6
7 # query the records from the RatiosAll table
8 table = conn.table("SELECT * FROM ratiosall")
9 # table is now a regular Picalo table, with all data
10 table.view()
11
12 # query the records to an iterator, this allows
13 # access to only one record at a time
14 results = conn.query("SELECT * FROM ratiosall")
15 for rec in results:
16     print rec.id
17
18 # close up shop
19 conn.close()

```

### 4.3 Modifying Data

Data can be modified in the source database using the `execute` method. This includes INSERT, UPDATE, CREATE TABLE, DROP, DELETE, and other data-modifying commands. These SQL commands do not return results, but simply modify records in the source database.

```

1 from picalo import *
2
3 # connect to the database
4 conn = Database.PostgreSQLConnection(database='fraudratios')
5
6 # delete a record
7 table = conn.execute("DELETE FROM RatiosAll WHERE id=3")
8 conn.commit()
9
10 # close up shop
11 conn.close()

```

**Posting Tables** Picalo includes a convenience function to post entire Dataset tables to databases. In other words, you can create a table using any method (such as `load_csv`) and then upload to the database. This is a convenience method provided to make importing of data from text files (CSV, TSV, etc.) and other sources into databases easy.

The method always tries to create a new table in the database – it will not append records into an existing table. If the table cannot be created (if

it exists, for example), an error is thrown. If the "replace" option is True, any existing tables by this name are dropped before the new table is created.

It uses the most common database types to create the database. It does not set any primary or secondary keys or referential integrity. This is a convenience function that is not possible to code in a perfectly portable way. Databases are simply too inconsistent. It will probably work for you, but may throw errors for some databases. Give it some testing before trusting it with your analyses.



# Chapter 5

## Scripting

Scripts are the reason Picalo was created. Many data analysis applications exist that perform different functions from their menu, but users who only know the graphical menu system of a program are at a serious disadvantage to scripts:

- Scripts provide automatic memory of the exact routine you performed. If you need to reperform a routine, scripts make it a simple matter of clicking the run button.
- Scripts can be improved upon over time. Most analysts redo the same routines from project to project. Scripts allow you to create a “personal toolbox” that you can take with you. Picalo itself grew out of a personal toolbox.
- Scripts can analyze thousands of transactions or even thousands of tables with a simple *for* loop. When you know scripting, you don’t need to sample anymore because you can write a script to look at 100 percent of the data.
- Scripts perform the analysis perfectly every time they are run (this is not to say you are going to write perfect routines, but the computer will perfectly follow your instructions every time :). Routines run using the menus are prone to human error in not following the specified steps.

Users are *highly* recommended to learn Picalo scripting (which uses Python). Everything in the Picalo interface is geared toward helping you learn how to program.

The Function Composer provides a graphical way to create Picalo commands. It should be useful to you as you learn how to script with Picalo.

## 5.1 The Shell

When you start Picalo, you'll immediately see the shell open at the bottom of the page. The shell allows you to type commands one at a time to produce results. In addition, the history page keeps track of everything you type so you can save sessions for later use.

In fact, everything you do with the Picalo menu and dialogs just translates to a call in the shell. As you navigate the program dialogs, watch the shell to see how Picalo is doing its work. Over time, you'll learn what commands do and become more familiar with the shell.

The interface and the shell are two-way, meaning that anything you change in the interface (such as modifying a cell value in a spreadsheet view) also changes in the shell variables. In like manner, anything you change in the shell variables is immediately reflected in the interface.

The shell is an excellent place to perform ad-hoc analyses or to test commands. Over time, it should become your primary interface with the Picalo engine.

## 5.2 Script Files

Script files are batched commands run in the shell. For example, you could write up an entire analysis in a script file, click run, and watch the entire routine run at once. For any formal analysis, you should write a script file. Script files provide history, allow maintenance, and create your toolbox for future efficiencies.

The examples directory within the Picalo documentation (also provided in the Appendix to this manual) give concrete examples of Picalo scripts in action.

Please see the Section 5.5 for information about the Python language.

## 5.3 Running Scripts

Once you create your script, run it in Picalo by selecting *Run Script* from the Script menu. Picalo has a *shared memory* model, which can be both a blessing and a curse. **It is important to understand this shared memory model.** This model means that variables are shared between the Picalo environment, the shell, and scripts you run. If a table is loaded into Picalo, you don't need to load it again in a script – the script can simply start modifying the table directly. In addition, any variables the script creates are available in Picalo *after* it finishes. This is extremely useful as it allows you to continue an analysis routine and to inspect variables after a script has processed.

The drawback of the shared memory model is you need to be careful not to override variables from one script to another. Normally, this isn't an issue because only one script runs at a time. However, if you set a table in a script, it will override any table set to that variable before the script runs.

You can only run one script at a time in a given Picalo window. If you have one script running and want to run another at the same time, select *Run Script in New Picalo*. This will start a new Picalo instance and run the script within it. This allows you to do two things at once, a useful option when you are running scripts that could take several hours to complete.

Running scripts in new Picalo instances is also useful when you want to start them fresh. Remember that all variables and tables create in a script you run stay in memory and are available in the shell afterward. Normally this is desirable so you can inspect your tables afterward, but sometimes the previous values of variables or tables can interact with new scripts you run and produce undesirable results.

## 5.4 Ideas for Successful Scripts

The following are ideas for writing successful scripts in Picalo:

- Read the Python Tutorial, as suggested in Section 5.5. You really need to understand **for** loops, **if** statements, variables, and other programming concepts to script effectively. This manual does not provide instruction on these topics because the Python Tutorial is well written.
- Place control totals throughout your scripts. Control totals are inter-

mediate values that you can use to check the accuracy of your script throughout its execution. Example control totals are number of records read from a file, totals of numerical values, and so forth.

- Place **print** statements before major sections of your code so you get feedback as it runs. Without print statements, you'll have to wait until the end of your script to know how it worked. Print statements are excellent places to place control totals.
- Code your scripts simply and in steps. Many people try to code multi-step processes without ever running their script. Instead, program one part of the analysis and run it; check your results and ensure it works. Then go onto the next step.
- Variables created in scripts remain active after the scripts finish. You can inspect tables and run extra commands as if those commands had been part of your script.
- Use the menu commands and Shell to test script lines. Every time you select a function from the menu, you'll see the command typed into the Shell for you. Everything typed into the Shell is placed in the Command Log. Use these features to interactively create scripts as you gain experience with the language.

## 5.5 The Python Language

The primary reason Picalo was built on Python is because Python is geared towards non-programmers and first-time users. Python is a beautiful, easy-to-learn language. Python programmers routinely produce efficient, quick, readable code. Yet, despite Python's simplistic approach, it is extremely powerful. It is used throughout the world as one of the primary scripting languages available.

Everything that Python offers is available in the shell and in your scripts. In practice, though, you only need to know a little bit about Python to use Picalo effectively. It is recommended that you read a few sections of the Python Tutorial at <http://docs.python.org/tut/tut.html> to familiarize yourself with the language as a primer. Focus on the following sections of the tutorial:

- Chapter 3: An informal introduction to Python
- Chapter 4: Flow Control Tools (if and for statements)

For most users, the tutorial may provide the information needed to write basic scripts. For readers who want more depth than the tutorial gives, the *Dive Into Python* web site at <http://www.diveintopython.org> provides even more information about Python. Focus on the “First Python Program” and “Native Datatypes” chapters. The section on regular expressions is also great for analysts.

Many other books on Python are available for purchase at your local or online bookstore. Most users will never need to understand Python in depth, but the full power of the language is available to those who want it.

## Chapter 6

# Picalo Reference

The functions in Picalo are grouped into categories, called modules. This section lists the individual modules with their functions.

Most functions are called in the format `Module.function()`. For example, to calculate a Benford's analysis, run `Benfords.analyze(...)`. The only exceptions are the core functions, which are called directly, as in `Table(...)` or `load_tsv(...)`.

Use this section as a reference to the functions in Picalo. This information is also available in two other places: 1) in the Function Composer (Edit menu), and 2) in the Shell window when a period is typed or a function is named.

# Chapter 7

## Core

The core functions in Picalo include the basic data types, Table manipulation, and loading/saving routines. Users should first read about the Table type, which is the foundation of everything done in Picalo. Almost all Picalo functions take a table as a parameter and return a Picalo table when done. This allows you to pass tables in and out of functions in your analysis routines.

### 7.1 boolean

Creates a new boolean object

**Signature:**

`boolean()`

### 7.2 check\_valid\_table

Checks to ensure the table is a valid table object, and that the columns are valid columns in the table. Throws an `AssertionError` if anything is wrong.

**Signature:**

`check_valid_table(<object> table)`

- **<object> table:**

## 7.3 clear\_progress

Clears the progress bar from the screen.

Sometimes multiple functions try to show or clear a progress bar. For example, a top-level script might show a master progress bar and then call `load()`. The `load()` function tries to show another progress bar, which Picalo normally circumvents or the `load()` function would take over the top-level script's progress bar. In other words, the first script to show a progress bar is the only one that can update and/or clear the dialog. By setting `force` to `True`, you can override this default behavior. This should not normally be used as it takes control when the top-level script should keep control.

### Signature:

```
clear_progress(<boolean> force=False)
```

- `<boolean> force` (optional): Whether to force control of the dialog to the calling code. If omitted, defaults to `False`.

## 7.4 count

Returns the number of items in the sequence. The built in function "len" also gives this value.

### Signature:

```
<float> = count(<list> sequence)
```

- `<list> sequence`: A sequence of items of any type. If not a sequence, returns 1.

Returns: The number of items in the sequence

## 7.5 currency.adjusted

Return the adjusted exponent of self

### Signature:

```
[currency].adjusted()
```



## 7.6 `currency.as_tuple`

Represents the number as a triple tuple.

To show the internals exactly as they are.

**Signature:**

```
[currency].as_tuple()
```

## 7.7 `currency.canonical`

Returns the same Decimal object.

As we do not have different encodings for the same number, the received object already is in its canonical form.

**Signature:**

```
[currency].canonical(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.8 `currency.compare`

Compares one to another.

`-1 => a < b` `0 => a = b` `1 => a > b` `NaN => one is NaN` Like `--cmp--`, but returns Decimal instances.

**Signature:**

```
[currency].compare(<object> other,  
                  <object> context=None)
```

- `<object> other`:
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.9 `currency.compare_signal`

Compares self to the other operand numerically.

It's pretty much like `compare()`, but all NaNs signal, with signaling NaNs taking precedence over quiet NaNs.

**Signature:**

```
[currency].compare_signal(<object> other,  
                          <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to `None`.

## 7.10 `currency.compare_total`

Compares self to other using the abstract representations.

This is not like the standard `compare`, which use their numerical value. Note that a total ordering is defined for all possible abstract representations.

**Signature:**

```
[currency].compare_total(<object> other)
```

- **<object> other:**

## 7.11 `currency.compare_total_mag`

Compares self to other using abstract repr., ignoring sign.

Like `compare_total`, but with operand's sign ignored and assumed to be 0.

**Signature:**

```
[currency].compare_total_mag(<object> other)
```

- **<object> other:**

## 7.12 `currency.conjugate`

None

**Signature:**

```
[currency].conjugate()
```

## 7.13 `currency.copy_abs`

Returns a copy with the sign set to 0.

**Signature:**

```
[currency].copy_abs()
```

## 7.14 `currency.copy_negate`

Returns a copy with the sign inverted.

**Signature:**

```
[currency].copy_negate()
```

## 7.15 `currency.copy_sign`

Returns self with the sign of other.

**Signature:**

```
[currency].copy_sign(<object> other)
```

- **<object> other:**

## 7.16 `currency.exp`

Returns `e ** self`.

**Signature:**

```
[currency].exp(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.17 `currency.fma`

Fused multiply-add.

Returns `self*other+third` with no rounding of the intermediate product `self*other`.

`self` and `other` are multiplied together, with no rounding of the result. The third operand is then added to the result, and a single final rounding is performed.

**Signature:**

```
[currency].fma(<object> other,  
               <object> third,  
               <object> context=None)
```

- `<object> other`:
- `<object> third`:
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.18 `currency.is_canonical`

Return `True` if `self` is canonical; otherwise return `False`.

Currently, the encoding of a `Decimal` instance is always canonical, so this method returns `True` for any `Decimal`.

**Signature:**

```
[currency].is_canonical()
```

## 7.19 `currency.is_finite`

Return True if self is finite; otherwise return False.

A Decimal instance is considered finite if it is neither infinite nor a NaN.

**Signature:**

```
[currency].is_finite()
```

## 7.20 `currency.is_infinite`

Return True if self is infinite; otherwise return False.

**Signature:**

```
[currency].is_infinite()
```

## 7.21 `currency.is_nan`

Return True if self is a qNaN or sNaN; otherwise return False.

**Signature:**

```
[currency].is_nan()
```

## 7.22 `currency.is_normal`

Return True if self is a normal number; otherwise return False.

**Signature:**

```
[currency].is_normal(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to None.

## 7.23 `currency.is_qnan`

Return True if self is a quiet NaN; otherwise return False.

**Signature:**

`[currency].is_qnan()`

## 7.24 `currency.is_signed`

Return True if self is negative; otherwise return False.

**Signature:**

`[currency].is_signed()`

## 7.25 `currency.is_snan`

Return True if self is a signaling NaN; otherwise return False.

**Signature:**

`[currency].is_snan()`

## 7.26 `currency.is_subnormal`

Return True if self is subnormal; otherwise return False.

**Signature:**

`[currency].is_subnormal(<object> context=None)`

- `<object> context` (optional): If omitted, defaults to None.

## 7.27 `currency.is_zero`

Return True if self is a zero; otherwise return False.

**Signature:**

`[currency].is_zero()`

## 7.28 `currency.ln`

Returns the natural (base e) logarithm of self.

**Signature:**

```
[currency].ln(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.29 `currency.log10`

Returns the base 10 logarithm of self.

**Signature:**

```
[currency].log10(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.30 `currency.logb`

Returns the exponent of the magnitude of self's MSD.

The result is the integer which is the exponent of the magnitude of the most significant digit of self (as though it were truncated to a single digit while maintaining the value of that digit and without limiting the resulting exponent).

**Signature:**

```
[currency].logb(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

### 7.31 `currency.logical_and`

Applies an 'and' operation between self and other's digits.

**Signature:**

```
[currency].logical_and(<object> other,  
                      <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.32 `currency.logical_invert`

Invert all its digits.

**Signature:**

```
[currency].logical_invert(<object> context=None)
```

- **<object> context (optional):** If omitted, defaults to None.

### 7.33 `currency.logical_or`

Applies an 'or' operation between self and other's digits.

**Signature:**

```
[currency].logical_or(<object> other,  
                    <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.



## 7.34 `currency.logical_xor`

Applies an 'xor' operation between self and other's digits.

**Signature:**

```
[currency].logical_xor(<object> other,  
                      <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.35 `currency.max`

Returns the larger value.

Like `max(self, other)` except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

**Signature:**

```
[currency].max(<object> other,  
              <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.36 `currency.max_mag`

Compares the values numerically with their sign ignored.

**Signature:**

```
[currency].max_mag(<object> other,  
                  <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.37 `currency.min`

Returns the smaller value.

Like `min(self, other)` except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

**Signature:**

```
[currency].min(<object> other,  
               <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.38 `currency.min_mag`

Compares the values numerically with their sign ignored.

**Signature:**

```
[currency].min_mag(<object> other,  
                  <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.39 `currency.next_minus`

Returns the largest representable number smaller than itself.

**Signature:**

```
[currency].next_minus(<object> context=None)
```

- **<object> context (optional):** If omitted, defaults to None.

## 7.40 `currency.next_plus`

Returns the smallest representable number larger than itself.

**Signature:**

```
[currency].next_plus(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.41 `currency.next_toward`

Returns the number closest to self, in the direction towards other.

The result is the closest representable number to self (excluding self) that is in the direction towards other, unless both have the same value. If the two operands are numerically equal, then the result is a copy of self with the sign set to be the same as the sign of other.

**Signature:**

```
[currency].next_toward(<object> other,  
                      <object> context=None)
```

- `<object> other`:
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.42 `currency.normalize`

Normalize- strip trailing 0s, change anything equal to 0 to 0e0

**Signature:**

```
[currency].normalize(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

### 7.43 `currency.number_class`

Returns an indication of the class of self.

The class is one of the following strings: sNaN NaN -Infinity -Normal -Subnormal -Zero +Zero +Subnormal +Normal +Infinity

**Signature:**

```
[currency].number_class(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to None.

### 7.44 `currency.quantize`

Quantize self so its exponent is the same as that of exp.

Similar to `self.rescale(exp._exp)` but with error checking.

**Signature:**

```
[currency].quantize(<object> exp,  
                   <object> rounding=None,  
                   <object> context=None,  
                   <object> watchexp=True)
```

- `<object> exp`:
- `<object> rounding` (optional): If omitted, defaults to None.
- `<object> context` (optional): If omitted, defaults to None.
- `<object> watchexp` (optional): If omitted, defaults to True.

### 7.45 `currency.radix`

Just returns 10, as this is Decimal, :)

**Signature:**

```
[currency].radix()
```

## 7.46 `currency remainder_near`

Remainder nearest to 0-  $\text{abs}(\text{remainder\_near}) \leq \text{other}/2$

**Signature:**

```
[currency].remainder_near(<object> other,  
                           <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.47 `currency.rotate`

Returns a rotated copy of self, value-of-other times.

**Signature:**

```
[currency].rotate(<object> other,  
                  <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.48 `currency.same_quantum`

Return True if self and other have the same exponent; otherwise return False.

If either operand is a special value, the following rules are used:

- return True if both operands are infinities
- return True if both operands are NaNs
- otherwise, return False.

**Signature:**

```
[currency].same_quantum(<object> other)
```

- **<object> other:**

## 7.49 `currency.scaleb`

Returns self operand after adding the second value to its exp.

**Signature:**

```
[currency].scaleb(<object> other,  
                 <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.50 `currency.shift`

Returns a shifted copy of self, value-of-other times.

**Signature:**

```
[currency].shift(<object> other,  
                <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.51 `currency.sqrt`

Return the square root of self.

**Signature:**

```
[currency].sqrt(<object> context=None)
```

- **<object> context (optional):** If omitted, defaults to None.

## 7.52 `currency.to_eng_string`

Convert to engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place.

Same rules for when in exponential and when as a value as in `__str__`.

**Signature:**

```
[currency].to_eng_string(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.53 `currency.to_integral`

Rounds to the nearest integer, without raising `Inexact`, rounded.

**Signature:**

```
[currency].to_integral(<object> rounding=None,  
                      <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to `None`.
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.54 `currency.to_integral_exact`

Rounds to a nearby integer.

If no rounding mode is specified, take the rounding mode from the context. This method raises the `Rounded` and `Inexact` flags when appropriate.

See also: `to_integral_value`, which does exactly the same as this method except that it doesn't raise `Inexact` or `Rounded`.

**Signature:**

```
[currency].to_integral_exact(<object> rounding=None,  
                             <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to `None`.
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.55 `currency.to_integral_value`

Rounds to the nearest integer, without raising `inexact`, rounded.

**Signature:**

```
[currency].to_integral_value(<object> rounding=None,  
                             <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to `None`.
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.56 `DateDelta`

A duration in time, such as the difference between two `DateTimes`. This is often used in `Grouping.stratify_by_date` and `Grouping.summarize_by_date`. Negative numbers specify durations that go backwards in time.

**Signature:**

```
<datetime.timedelta> = DateDelta(<int,  
                                  long,  
                                  or float> days=0,  
                                  <int,  
                                  long,  
                                  or float> seconds=0,  
                                  <int,  
                                  long,  
                                  or float> microseconds=0,  
                                  <int,  
                                  long,  
                                  or float> milliseconds=0,  
                                  <int,  
                                  long,  
                                  or float> minutes=0,  
                                  <int,  
                                  long,  
                                  or float> hours=0,  
                                  <int,
```



```
long,  
or float> weeks=0)
```

- <int, long, or float> days (optional): The number of days in this duration. If omitted, defaults to 0.
- <int, long, or float> seconds (optional): The number of seconds in this duration. If omitted, defaults to 0.
- <int, long, or float> microseconds (optional): The number of microseconds in this duration. If omitted, defaults to 0.
- <int, long, or float> milliseconds (optional): The number of milliseconds in this duration. If omitted, defaults to 0.
- <int, long, or float> minutes (optional): The number of minutes in this duration. If omitted, defaults to 0.
- <int, long, or float> hours (optional): The number of hours in this duration. If omitted, defaults to 0.
- <int, long, or float> weeks (optional): The number of weeks in this duration. If omitted, defaults to 0.

Returns: A `datetime.timedelta` object

## 7.57 DateFormatter

Formats a `DateTime` or `Date` object for printing using the given format. Search the web for "strftime unix manpage" for the formatting tokens.

### Signature:

```
DateFormat(<object> datetime_value,  
           <object> format=None)
```

- <object> `datetime_value`:
- <object> `format` (optional): If omitted, defaults to `None`.

## 7.58 DateTimeFormat

Formats a DateTime or Date object for printing using the given format. Search the web for "strftime unix manpage" for the formatting tokens.

### Signature:

```
DateTimeFormat(<object> datetime_value,  
               <object> format=None)
```

- **<object> datetime\_value:**
- **<object> format (optional):** If omitted, defaults to None.

## 7.59 error

An error type that signifies that an error occurred when calculating the value for some cell. Don't create this directly. Picalo creates them when errors are encountered during analyses so the analysis doesn't get stopped by small errors.

The error object is useful in that it encodes information about what went wrong. Although it only prints as <err> in table listings, it can be inspected as shown in the set\_type method example.

### Signature:

```
error(<Exception> exc=None,  
      <str> msg="None",  
      <object> previous=None)
```

- **<Exception> exc (optional):** The exception object. If omitted, defaults to None.
- **<str> msg (optional):** Allows you to provide a specific error message. Defaults to str(exception). If omitted, defaults to None.
- **<object> previous (optional):** The previous value of the cell, so we don't lose it. If omitted, defaults to None.

## 7.60 `error.get_exception`

Returns the exception object associated with this error. This might be `None` if no object was created.

**Signature:**

```
<Exception> = [error].get_exception()
```

Returns: The exception object that caused this error.

## 7.61 `error.get_message`

Returns the actual error message. This allows the object to be interrogated for what really went wrong.

**Signature:**

```
<str> = [error].get_message()
```

Returns: The error message.

## 7.62 `error.get_previous`

Returns the previous value of the cell before the error was placed in the table. This might be `none` if the error didn't replace a cell value.

**Signature:**

```
<object> = [error].get_previous()
```

Returns: The previous value of the cell where this error occurred.

## 7.63 load

Loads a native picalo file. Determines the version of the picalo file, then calls the appropriate load routine for that version.

### Signature:

```
<Table or TableList or TableArray> = load(<str> filename)
```

- **<str> filename:** The name of the file to load. This can also be an open stream.

Returns: A Picalo table containing the data from the file.

## 7.64 load\_csv

Loads records from a Comma Separated Values (CSV) file. Among the various CSV flavors in the world, this function uses the Microsoft version.

### Signature:

```
<Table> = load_csv(<str> filename,  
                  <str> header_row="True",  
                  <str> none="",  
                  <str> encoding="None",  
                  <str> errors="replace")
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> header\_row (optional):** Whether the first line of the file includes the column names. Most delimited text files do this. If omitted, defaults to True.
- **<str> none (optional):** An optional argument that specifies the record to assign Python's None type to (for blank cells). If omitted, defaults to .

- `<str> encoding` (optional): The unicode encoding to write with. This should be a value from the `codecs` module. If `None`, the encoding is guessed to `utf-8`, `utf-16`, `utf-16-be`, or `utf-16-le`. If omitted, defaults to `None`.
- `<str> errors` (optional): How to handle characters that cannot be handled. Options are `'replace'`, `'strict'`, and `'ignore'`. See `'codecs'` in the Python documentation for more information. If omitted, defaults to `replace`.

Returns: A new `Table` object, or `None` if the file is empty.

## 7.65 load\_delimited

Loads records from the given delimited text file. This function allows the specification of delimiters and qualifiers. Most users should use the `load_csv`, `load_tsv`, and `load_fixed` functions as they provide easier access to text files. Use this function only if you have a specially-formatted text file.

### Signature:

```
<Table> = load_delimited(<str> filename,
                        <str> header_row="True",
                        <str> delimiter=",",
                        <str> qualifier="\"",
                        <str> none="",
                        <str> encoding="None",
                        <str> errors="replace")
```

- **<str> filename:** A file name or a file pointer to an open file.
- `<str> header_row` (optional): Whether the first line of the file includes the column names. Most delimited text files do this. If omitted, defaults to `True`.
- `<str> delimiter` (optional): An optional field delimiter character. If omitted, defaults to `.,`

- `<str>` **qualifier** (optional): An optional qualifier to use when delimiters exist in field records. If omitted, defaults to `"`.
- `<str>` **none** (optional): An optional argument that specifies the record to assign Python's `None` type to (for blank cells). If omitted, defaults to `.`
- `<str>` **encoding** (optional): The unicode encoding to write with. This should be a value from the `codecs` module. If `None`, the encoding is guessed to `utf-8`, `utf-16`, `utf-16-be`, or `utf-16-le`. If omitted, defaults to `None`.
- `<str>` **errors** (optional): How to handle characters that cannot be handled. Options are `'replace'`, `'strict'`, and `'ignore'`. See `'codecs'` in the Python documentation for more information. If omitted, defaults to `replace`.

Returns: A new `Table` object, or `None` if the file is empty.

## 7.66 load\_excel

Loads the given Excel file. Values are taken from the first sheet in the workbook.

### Signature:

```
<Table> = load_excel(<str> filename,
                    <str> worksheet_name="None",
                    <str> topleft_cell="None",
                    <str> bottomright_cell="None",
                    <str> header_row="True",
                    <str> none="")
```

- `<str>` **filename**: The file name. It must be the string name and not a file pointer (the library doesn't support it).
- `<str>` **worksheet\_name** (optional): The name of the worksheet that contains the data (only one worksheet can be imported). Defaults to the first sheet in the workbook. If omitted, defaults to `None`.

- `<str> topleft_cell` (optional): The top-left cell of the block to import. It should be in spreadsheet format, such as "A2" or "C5". Defaults to the first cell with data. If omitted, defaults to None.
- `<str> bottomright_cell` (optional): The bottom-right cell of the block to import. It should be in spreadsheet format, such as "A2" or "C5". Defaults to the last cell with data. If omitted, defaults to None.
- `<str> header_row` (optional): Whether the first line of the file includes the column names. Defaults to True. If omitted, defaults to True.
- `<str> none` (optional): An optional argument that specifies the record to assign Python's None type to (for blank cells). If omitted, defaults to .

Returns: A new Table object, or None if the file is empty.

## 7.67 load\_fixed

Loads records from a fixed width file. Fixed width files pad columns with extra spaces so they are easy to read with a text editor.

### Signature:

```
<Table> = load_fixed(<str> filename,
                    <list> column_positions=[],
                    <bool> header_row=True,
                    <str> none="",
                    <str> encoding="None",
                    <str> errors="replace",
                    <bool> line_separators=True)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<list> column_positions` (optional): A list of integers specifying where to split columns. The first item should always be 0 (the first character in each line). The last item should be the ending column position. If omitted, defaults to [].

- `<bool> header_row` (optional): An optional parameter to specify whether the first row of the file contains field headers. Defaults to `True`. If omitted, defaults to `True`.
- `<str> none` (optional): An optional argument that specifies the record to assign Python's `None` type to (for blank cells). If omitted, defaults to `.`
- `<str> encoding` (optional): The unicode encoding to read with. This should be a value from the `codecs` module. If `None`, the encoding is guessed to `utf_8`, `utf-16`, `utf-16-be`, or `utf-16-le`. If omitted, defaults to `None`.
- `<str> errors` (optional): How to handle characters that cannot be handled. Options are `'replace'`, `'strict'`, and `'ignore'`. See `'codecs'` in the Python documentation for more information. If omitted, defaults to `replace`.
- `<bool> line_separators` (optional): Whether the lines in the file are separated by hard returns. See the examples above for more information. If omitted, defaults to `True`.

Returns: A new `Table` object, or `None` if the file is empty.

### Example 1:

```

1
2 Suppose test-fixed.txt contains the following data:
3 id name salary
4 1 Marge 50000
5 2 Dan 4500
6
7 >>> # load the data into a new table, specifying column positions as 0-2, 2-8, and 8-15
8 >>> data2 = load_fixed('test-fixed.txt', [0,2,8,15])
9 >>> data2.view()
10 +-----+
11 | id | name | salary |
12 +-----+
13 | 1 | Marge | 50000 |
14 | 2 | Dan | 4500 |
15 +-----+
16 The next step is to convert column types as appropriate.
```

### Example 2:



```

1
2 Suppose test-fixed.txt contains the following data:
3 idname salary01Marge05000002Dan 004500
4 Note that in this file, the lines are not separated by hard returns. This is common in mainframe
5 data files like EBCDIC-encoded files.
6
7 >>> # load the data into a new table, specifying column positions as 0-2, 2-7, and 7-13
8 >>> data2 = load_fixed('test-fixed.txt', [0,2,7,13], line_separators=False)
9 >>> data2.view()
10 +-----+
11 | id | name | salary |
12 +-----+
13 | 01 | Marge | 050000 |
14 | 02 | Dan | 004500 |
15 +-----+
16 The next step is to convert column types as appropriate.

```

## 7.68 load\_tsv

Loads records from a Excel Tab Separated Values (TSV) file. Among the various TSV flavors in the world, this function uses the Microsoft version.

### Signature:

```

<Table> = load_tsv(<str> filename,
                   <str> header_row="True",
                   <str> none="",
                   <str> encoding="None",
                   <str> errors="replace")

```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> header\_row (optional):** Whether the first line of the file includes the column names. Most delimited text files do this. If omitted, defaults to True.
- **<str> none (optional):** An optional argument that specifies the record to assign Python's None type to (for blank cells). If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to write with. This should be a value from the codecs module. If None, the encoding is

guessed to utf-8, utf-16, utf-16-be, or utf-16-le. If omitted, defaults to None.

- `<str> errors` (optional): How to handle characters that cannot be handled. Options are 'replace', 'strict', and 'ignore'. See 'codecs' in the Python documentation for more information. If omitted, defaults to replace.

Returns: A new Table object, or None if the file is empty.

## 7.69 max

With a single sequence argument, return its largest item. With two or more arguments, return the largest argument.

**Signature:**

```
<object> = max()
```

Returns: The largest item in the sequence

## 7.70 mean

Returns the average of the given sequence, or the default if the sequence is empty. More advanced statistical routines can be found in the `picalo.lib.stats` module.

**Signature:**

```
<float> = mean(<list> sequence,  
              <int> default=0)
```

- **<list> sequence:** A sequence of numbers
- **<int> default** (optional): The default value to return when the list is empty. If omitted, defaults to 0.

Returns: The average of the numbers

## 7.71 min

With a single sequence argument, return its smallest item. With two or more arguments, return the smallest argument.

**Signature:**

```
<object> = min()
```

Returns: The smallest item in the sequence

## 7.72 number.adjusted

Return the adjusted exponent of self

**Signature:**

```
[number].adjusted()
```

## 7.73 number.as\_tuple

Represents the number as a triple tuple.

To show the internals exactly as they are.

**Signature:**

```
[number].as_tuple()
```

## 7.74 number.canonical

Returns the same Decimal object.

As we do not have different encodings for the same number, the received object already is in its canonical form.

**Signature:**

```
[number].canonical(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

## 7.75 `number.compare`

Compares one to another.

-1 => a < b 0 => a = b 1 => a > b NaN => one is NaN Like `--cmp--`, but returns Decimal instances.

**Signature:**

```
[number].compare(<object> other,  
                 <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.76 `number.compare_signal`

Compares self to the other operand numerically.

It's pretty much like `compare()`, but all NaNs signal, with signaling NaNs taking precedence over quiet NaNs.

**Signature:**

```
[number].compare_signal(<object> other,  
                        <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.77 `number.compare_total`

Compares self to other using the abstract representations.

This is not like the standard `compare`, which use their numerical value. Note that a total ordering is defined for all possible abstract representations.

**Signature:**

```
[number].compare_total(<object> other)
```

- **<object> other:**

## 7.78 `number.compare_total_mag`

Compares self to other using abstract repr., ignoring sign.

Like `compare_total`, but with operand's sign ignored and assumed to be 0.

**Signature:**

```
[number].compare_total_mag(<object> other)
```

- `<object> other`:

## 7.79 `number.conjugate`

None

**Signature:**

```
[number].conjugate()
```

## 7.80 `number.copy_abs`

Returns a copy with the sign set to 0.

**Signature:**

```
[number].copy_abs()
```

## 7.81 `number.copy_negate`

Returns a copy with the sign inverted.

**Signature:**

```
[number].copy_negate()
```

## 7.82 `number.copy_sign`

Returns self with the sign of other.

**Signature:**

```
[number].copy_sign(<object> other)
```

- **<object> other:**

## 7.83 `number.exp`

Returns  $e^{** self}$ .

**Signature:**

```
[number].exp(<object> context=None)
```

- **<object> context (optional):** If omitted, defaults to None.

## 7.84 `number.fma`

Fused multiply-add.

Returns  $self * other + third$  with no rounding of the intermediate product  $self * other$ .

self and other are multiplied together, with no rounding of the result. The third operand is then added to the result, and a single final rounding is performed.

**Signature:**

```
[number].fma(<object> other,  
             <object> third,  
             <object> context=None)
```

- **<object> other:**
- **<object> third:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.85 `number.is_canonical`

Return True if self is canonical; otherwise return False.

Currently, the encoding of a Decimal instance is always canonical, so this method returns True for any Decimal.

**Signature:**

```
[number].is_canonical()
```

## 7.86 `number.is_finite`

Return True if self is finite; otherwise return False.

A Decimal instance is considered finite if it is neither infinite nor a NaN.

**Signature:**

```
[number].is_finite()
```

## 7.87 `number.is_infinite`

Return True if self is infinite; otherwise return False.

**Signature:**

```
[number].is_infinite()
```

## 7.88 `number.is_nan`

Return True if self is a qNaN or sNaN; otherwise return False.

**Signature:**

```
[number].is_nan()
```

## 7.89 `number.is_normal`

Return True if self is a normal number; otherwise return False.

**Signature:**

```
[number].is_normal(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to None.

## 7.90 `number.is_qnan`

Return True if self is a quiet NaN; otherwise return False.

**Signature:**

```
[number].is_qnan()
```

## 7.91 `number.is_signed`

Return True if self is negative; otherwise return False.

**Signature:**

```
[number].is_signed()
```

## 7.92 `number.is_snan`

Return True if self is a signaling NaN; otherwise return False.

**Signature:**

```
[number].is_snan()
```



### 7.93 `number.is_subnormal`

Return True if self is subnormal; otherwise return False.

**Signature:**

```
[number].is_subnormal(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

### 7.94 `number.is_zero`

Return True if self is a zero; otherwise return False.

**Signature:**

```
[number].is_zero()
```

### 7.95 `number.ln`

Returns the natural (base e) logarithm of self.

**Signature:**

```
[number].ln(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

### 7.96 `number.log10`

Returns the base 10 logarithm of self.

**Signature:**

```
[number].log10(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

## 7.97 number.logb

Returns the exponent of the magnitude of self's MSD.

The result is the integer which is the exponent of the magnitude of the most significant digit of self (as though it were truncated to a single digit while maintaining the value of that digit and without limiting the resulting exponent).

### Signature:

```
[number].logb(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

## 7.98 number.logical\_and

Applies an 'and' operation between self and other's digits.

### Signature:

```
[number].logical_and(<object> other,  
                    <object> context=None)
```

- <object> other:
- <object> context (optional): If omitted, defaults to None.

## 7.99 number.logical\_invert

Invert all its digits.

### Signature:

```
[number].logical_invert(<object> context=None)
```

- <object> context (optional): If omitted, defaults to None.

## 7.100 `number.logical_or`

Applies an 'or' operation between self and other's digits.

**Signature:**

```
[number].logical_or(<object> other,  
                   <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.101 `number.logical_xor`

Applies an 'xor' operation between self and other's digits.

**Signature:**

```
[number].logical_xor(<object> other,  
                   <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.102 `number.max`

Returns the larger value.

Like `max(self, other)` except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

**Signature:**

```
[number].max(<object> other,  
            <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.103 `number.max_mag`

Compares the values numerically with their sign ignored.

**Signature:**

```
[number].max_mag(<object> other,  
                 <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.104 `number.min`

Returns the smaller value.

Like `min(self, other)` except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

**Signature:**

```
[number].min(<object> other,  
             <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

### 7.105 `number.min_mag`

Compares the values numerically with their sign ignored.

**Signature:**

```
[number].min_mag(<object> other,  
                 <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.106 `number.next_minus`

Returns the largest representable number smaller than itself.

**Signature:**

```
[number].next_minus(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.107 `number.next_plus`

Returns the smallest representable number larger than itself.

**Signature:**

```
[number].next_plus(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.108 `number.next_toward`

Returns the number closest to self, in the direction towards other.

The result is the closest representable number to self (excluding self) that is in the direction towards other, unless both have the same value. If the two operands are numerically equal, then the result is a copy of self with the sign set to be the same as the sign of other.

**Signature:**

```
[number].next_toward(<object> other,  
                    <object> context=None)
```

- `<object> other`:
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.109 `number.normalize`

Normalize- strip trailing 0s, change anything equal to 0 to 0e0

**Signature:**

```
[number].normalize(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to None.

## 7.110 `number.number_class`

Returns an indication of the class of self.

The class is one of the following strings: sNaN NaN -Infinity -Normal -Subnormal -Zero +Zero +Subnormal +Normal +Infinity

**Signature:**

```
[number].number_class(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to None.

## 7.111 `number.quantize`

Quantize self so its exponent is the same as that of exp.

Similar to `self._rescale(exp._exp)` but with error checking.

**Signature:**

```
[number].quantize(<object> exp,  
                  <object> rounding=None,  
                  <object> context=None,  
                  <object> watchexp=True)
```

- `<object> exp`:
- `<object> rounding` (optional): If omitted, defaults to None.
- `<object> context` (optional): If omitted, defaults to None.
- `<object> watchexp` (optional): If omitted, defaults to True.

## 7.112 `number.radix`

Just returns 10, as this is Decimal, :)

**Signature:**

```
[number].radix()
```

## 7.113 `number.remainder_near`

Remainder nearest to 0-  $\text{abs}(\text{remainder\_near}) \leq \text{other}/2$

**Signature:**

```
[number].remainder_near(<object> other,  
                        <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.114 `number.rotate`

Returns a rotated copy of self, value-of-other times.

**Signature:**

```
[number].rotate(<object> other,  
               <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.115 `number.same_quantum`

Return True if self and other have the same exponent; otherwise return False.

If either operand is a special value, the following rules are used:

- return True if both operands are infinities
- return True if both operands are NaNs
- otherwise, return False.

**Signature:**

```
[number].same_quantum(<object> other)
```

- **<object> other:**

## 7.116 `number.scaleb`

Returns self operand after adding the second value to its exp.

**Signature:**

```
[number].scaleb(<object> other,  
                <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.

## 7.117 `number.shift`

Returns a shifted copy of self, value-of-other times.

**Signature:**

```
[number].shift(<object> other,  
               <object> context=None)
```

- **<object> other:**
- **<object> context (optional):** If omitted, defaults to None.



## 7.118 `number.sqrt`

Return the square root of self.

### Signature:

```
[number].sqrt(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.119 `number.to_eng_string`

Convert to engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place.

Same rules for when in exponential and when as a value as in `__str__`.

### Signature:

```
[number].to_eng_string(<object> context=None)
```

- `<object> context` (optional): If omitted, defaults to `None`.

## 7.120 `number.to_integral`

Rounds to the nearest integer, without raising `inexact`, rounded.

### Signature:

```
[number].to_integral(<object> rounding=None,  
                    <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to `None`.
- `<object> context` (optional): If omitted, defaults to `None`.

## 7.121 `number.to_integral_exact`

Rounds to a nearby integer.

If no rounding mode is specified, take the rounding mode from the context. This method raises the Rounded and Inexact flags when appropriate.

See also: `to_integral_value`, which does exactly the same as this method except that it doesn't raise Inexact or Rounded.

### Signature:

```
[number].to_integral_exact(<object> rounding=None,  
                           <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to None.
- `<object> context` (optional): If omitted, defaults to None.

## 7.122 `number.to_integral_value`

Rounds to the nearest integer, without raising inexact, rounded.

### Signature:

```
[number].to_integral_value(<object> rounding=None,  
                           <object> context=None)
```

- `<object> rounding` (optional): If omitted, defaults to None.
- `<object> context` (optional): If omitted, defaults to None.

## 7.123 `save_csv`

Saves this table to a Comma Separated Values (CSV) text file. CSV is an industry-standard way of transferring data between applications. This is the preferred way of exporting data from Picalo.

Note that although this is the preferred export method, it has some limitations. These are limitations of the format rather than limitations of Picalo:

- No type information is saved to the file. All data is essentially turned into strings.
- Be sure to use the correct encoding if using international languages.

- Different standards for CSV exist (that's the nice thing about standards :). This export uses the Microsoft version.

Note that Microsoft Office seems to like CSV files better than TSV files.

### Signature:

```
save_csv(<Table> table,  
        <str> filename,  
        <str> line_ending="",  
        <str> none="",  
        <str> encoding="utf-8",  
        <object> respect_filter=False)
```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with. If omitted, defaults to .
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.124 save\_delimited

Saves this table to a delimited text file. This method allows the specification of different types of delimiters and qualifiers.

This method is for advanced users. Most users should call `save_tsv`, `save_csv`, or `save_fixed` to save using one of the accepted text formats. See these methods for more information.

**Signature:**

```
save_delimited(<Table> table,
               <str> filename,
               <str> delimiter=",",
               ",
               <str> qualifier="\"",
               <str> line_ending="
",
               <str> none="",
               <str> encoding="utf-8",
               <object> respect_filter=False)
```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> delimiter (optional):** A field delimiter character, defaults to a comma (,) If omitted, defaults to ,.
- **<str> qualifier (optional):** A qualifier to use when delimiters exist in field records, defaults to a double quote (") If omitted, defaults to ".
- **<str> line\_ending (optional):** A line ending to separate rows with, defaults to `os.linesep` ( If omitted, defaults to .
- **<str> none (optional):** An parameter specifying what to write for cells that have the `None` value, defaults to an empty string (") If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to write with. This should be a value from the `codecs` module, defaults to 'utf-8'. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to `False`.

## 7.125 save\_excel

Saves this table to a Microsoft Excel 97+ file.

**Signature:**

```
save_excel(<Table> table,
          <str> filename,
          <str> none="",
          <object> respect_filter=False)
```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to .
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.126 save\_fixed

Saves this table to a fixed width text file. Fixed width files pad column records with extra spaces so they are easier to read.

You should normally prefer CSV and TSV export formats to fixed as they have less limitations. Fixed format was widely used in the early days of computers, and some servers still use the format.

```
>>> # create a table and save it >>> data = Table([ >>> ( 'id',
int ), >>> ( 'name', unicode ), >>> ( 'salary', number ), >>> ]) >>>
data.append(1, 'Marge', 50000) >>> data.append(2, 'Danny', 45000) >>>
data.save_fixed('test-fixed.txt')
```

Test-fixed.txt now contains the following data: id name salary 1 Marge 50000 2 Danny 45000

**Signature:**

```

save_fixed(<Table> table,
          <str> filename,
          <str> line_ending="
",
          <str> none="",
          <object> encoding=utf-8,
          <object> respect_filter=False)

```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with. If omitted, defaults to `.`
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to `.`
- **<object> encoding (optional):** If omitted, defaults to `utf-8`.
- **<object> respect\_filter (optional):** If omitted, defaults to `False`.

## 7.127 save\_tsv

Saves this table to a Tab Separated Values (TSV) text file. TSV is an industry-standard way of transferring data between applications.

Note that although this is the preferred export method, it has some limitations. These are limitations of the format rather than limitations of Picalo:

- No type information is saved to the file. All data is essentially turned into strings.
- Be sure to use the correct encoding if using international languages.
- Different standards for TSV exist (that's the nice thing about standards :).

This export uses the Microsoft version.

Note that Microsoft Office seems to like CSV files better than TSV files.

### Signature:

```

save_tsv(<Table> table,
        <str> filename,

```

```

        <str> line_ending="
",
        <str> none="",
        <str> encoding="utf-8",
        <object> respect_filter=False)

```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with If omitted, defaults to .
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.128 save\_xml

Saves this table to an XML file using a pre-defined schema. If you need to save to a different schema, use the `xml.dom.minidom` class directly.

### Signature:

```

save_xml(<Table> table,
        <str> filename,
        <str> line_ending="
",
        <str> indent=" ",
        <bool> compact=False,
        <str> none="",

```

```
<object> encoding=utf-8,  
<object> respect_filter=False)
```

- **<Table> table:** The table to save
- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with (when compact is False) If omitted, defaults to .
- **<str> indent (optional):** The character(s) to use for indenting (when compact is False) If omitted, defaults to .
- **<bool> compact (optional):** Whether to compact the XML or make it "pretty" with whitespace If omitted, defaults to False.
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to .
- **<object> encoding (optional):** If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.129 show\_progress

Updates the progress bar with a message and a percentage progress between 0 and 1. To remove the progress bar, call `clear_progress()`.

This function is important because it gives feedback to the user. In addition, and perhaps more importantly, it gives the user a cancel button (in GUI mode) that allows the user to cancel your script. Be sure to call `show_progress` throughout your script.

Sometimes multiple functions try to show or clear a progress bar. For example, a top-level script might show a master progress bar and then call `load()`. The `load()` function tries to show another progress bar, which Picalo normally circumvents or the `load()` function would take over the top-level script's progress bar. In other words, the first script to show a progress bar is the only one that can update and/or clear the dialog. By setting `force` to



True, you can override this default behavior. This should not normally be used as it takes control when the top-level script should keep control.

**Signature:**

```
show_progress(<str> msg="",  
              <float> progress=1.0,  
              <str> title="Progress",  
              <boolean> force=False)
```

- **<str> msg (optional):** The message to show the user. If omitted, defaults to .
- **<float> progress (optional):** A value between 0 and 1 indicating the percentage finished. If omitted, defaults to 1.0.
- **<str> title (optional):** The title of the progress bar. Defaults to 'Progress'. If omitted, defaults to Progress.
- **<boolean> force (optional):** Whether to force control of the dialog to the calling code. If omitted, defaults to False.

## 7.130 stdev

Returns the standard deviation of the given sequence, or the default if the sequence contains zero or one items. More advanced statistical routines can be found in the `picalo.lib.stats` module.

**Signature:**

```
<float> = stdev(<list> sequence,  
               <float> default=0)
```

- **<list> sequence:** A sequence of items.
- **<float> default (optional):** The default if a standard deviation cannot be calculated. If omitted, defaults to 0.

Returns: The standard deviation of the sequence, or the default if `len(sequence) < 2`.

## 7.131 sum

Returns the sum of the given sequence of numbers plus the value of start. When the sequence is empty, returns start.

### Signature:

```
<int> = sum(<list> sequence,
           <int> start=0)
```

- **<list> sequence:** A sequence of numbers
- **<int> start (optional):** The starting value, usually 0. If omitted, defaults to 0.

Returns: The sum of the sequence, plus the start value

## 7.132 Table

Creates a Picalo Table. Tables are the single-most important item in Picalo. It should be the first place users look to understand how to use the program. See the manual for more information on how tables work.

The columns (field definitions) are required to be named and to have types. Fields are specified as a sequence of (name, type) pairs. The name can be any string starting with a letter and then any combination of letters and numbers. Column names must be unique. The type must be a type object, such as int, float, unicode, DateTime, etc. See below for examples.

The initial data for the table can be specified as a sequence of sequences, or a grid of data. See below for examples.

### Signature:

```
<Table> = Table(<list> columns=3,
               <Table/list> data=[])
```

- **<list> columns (optional):** A list of (name, type) pairs specifying the column names and their types. If omitted, defaults to 3.
- **<Table/list> data (optional):** The initial data for the table specified as another Picalo table or a list of lists. If omitted, defaults to [].

Returns: The new Picalo table.

### Example 1:

```

1 >>> # creates a table with two columns
2 >>> data = Table([
3 ...   ( 'id',   int ),
4 ...   ( 'name', unicode),
5 ... ])
6 >>> data.structure().view()
7 +-----+-----+-----+-----+
8 | Column |      Type      | Expression | Format |
9 +-----+-----+-----+-----+
10 | id     | <type 'int '>  | <N>       | <N>   |
11 | name   | <type 'unicode '> | <N>       | <N>   |
12 +-----+-----+-----+-----+

```

### Example 2:

```

1 >>> # creates a table with two columns and initial data
2 >>> data = Table([
3 ...   ( 'id',   int ),
4 ...   ( 'name', unicode),
5 ... ], [
6 ...   [ 1, 'Danny' ],
7 ...   [ 2, 'Vijay' ],
8 ...   [ 3, 'Dongsong' ],
9 ...   [ 4, 'Sally' ],
10 ... ])
11 >>> data.structure().view()
12 +-----+-----+-----+-----+
13 | Column |      Type      | Expression | Format |
14 +-----+-----+-----+-----+
15 | id     | <type 'int '>  | <N>       | <N>   |
16 | name   | <type 'unicode '> | <N>       | <N>   |
17 +-----+-----+-----+-----+

```

### Example 3:

```

1 >>> # creates a table with two columns (using shortcut notation)
2 >>> # since types are not specified, both columns are typed as unicode
3 >>> data = Table(['id', 'name'])
4 >>> data.structure().view()
5 +-----+-----+-----+-----+
6 | Column |      Type      | Expression | Format |
7 +-----+-----+-----+-----+
8 | id     | <type 'unicode '> | <N>       | <N>   |
9 | name   | <type 'unicode '> | <N>       | <N>   |
10 +-----+-----+-----+-----+

```

### Example 4:

```

1 >>> # creates a table with two columns (using another shortcut notation)
2 >>> data = Table(3)
3 >>> data.structure().view()
4 +-----+-----+-----+-----+

```

5	Column	Type	Expression	Format
6				
7	col000	<type 'unicode '>	<N>	<N>
8	col001	<type 'unicode '>	<N>	<N>
9	col002	<type 'unicode '>	<N>	<N>
10				

### 7.133 Table.append

Inserts a new record at the end of this table. This is the primary way to add new data to a table. If you need to insert a row in the middle of a table, use the insert() method.

Records can be added in any of the following ways:

Format 1: - newrec = mytable3.append() - newrec['ID'] = 4 - newrec['Name'] = 'Homer' - newrec['Salary'] = 15000

Format 2: - mytable.append(4, 'Homer', 1500)

Format 3: - mytable3.append([4, 'Homer', 1500])

Format 4: - mytable3.append({'ID':5, 'Name':'Marge', 'Salary': 275000})

Format 5: - mytable3.append({0:5, 1:'Marge', 2: 275000})

Format 6: - mytable3.append(ID=5, Name='Krusty', Salary=50000)

You cannot mix formats in the same call.

#### Signature:

<Record> = [Table].append()

Returns: The new record object (that was appended to the end of the table)

### 7.134 Table.append\_calculated

Adds a new, calculated column with records given by expression. Calculated columns act as regular columns in all ways. Their records are 'active' meaning their records change when the result of the expression. In other words, they are recalculated each time they are used rather than being stored statically. This is similar to the way Excel functions always reflect the most updated data records.

#### Signature:

```
<Column> = [Table].append_calculated(<str> name,
                                     <str> expression)
```

- **<str> name:** The new column name
- **<str> expression:** An expression that returns the record of the new field. As shown in the example, use `rec` to denote the current record being evaluated.

Returns: The new column object.

#### Example:

```
1 >>> table = Table([('id', int)], [[1],[2],[4]])
2 >>> table.append_calculated('plusone', "col000+1")
3 >>> table.view()
4 +-----+
5 | col000 | plusone |
6 +-----+
7 |      1 |       2 |
8 |      2 |       3 |
9 |      4 |       5 |
10 +-----+
```

### 7.135 Table.append\_calculated\_static

Adds a new, calculated column with records given by expression. The records are calculated immediately using expression, and then they are static. In other words, this method calculates a new, regular column. The records are not 'active' in the sense that `append_calculated()` columns are active. When this method returns, the new column is the same as any other, non-calculated column.

#### Signature:

```
<Column> = [Table].append_calculated_static(<str> name,
                                           <type> column_type,
                                           <str> expression)
```

- **<str> name:** The new column name
- **<type> column\_type:** The type of the new column (str, Date, int, long, etc.)

- **<str> expression:** An expression that returns the record of the new field. As shown in the example, use `rec` to denote the current record being evaluated.

Returns: The new column object.

#### Example:

```

1 >>> table = Table([('id', int)], [[1],[2],[4]])
2 >>> table.append_calculated('plusone', int, "col000+1")
3 >>> table.view()
4 +-----+
5 | col000 | plusone |
6 +-----+
7 |      1 |       2 |
8 |      2 |       3 |
9 |      4 |       5 |
10 +-----+

```

## 7.136 Table.append\_column

Adds a new column to the table, optionally setting records of the new cells.

#### Signature:

```

<Column> = [Table].append_column(<str> name,
                                   <type> column_type,
                                   <list,
                                   Column,
                                   or Table> records=None)

```

- **<str> name:** The new column name
- **<type> column\_type:** The new column type (int, float, DateTime, unicode, str, etc)
- **<list, Column, or Table> records (optional):** A list of records to place into the cells of the new column (if a full Table, the first column is used) If omitted, defaults to None.

Returns: The new column object.

### 7.137 `Table.clear_filter`

Clears any active filter on this table, restoring the view to all records in the table

**Signature:**

```
[Table].clear_filter()
```

### 7.138 `Table.column`

Returns a single column of the table. Column records can be read and modified but not deleted. This is a useful method when you want to work with a single column of a table.

The returned Column object is essentially a list of records and can be treated as such. Any function that takes a list can take a Column object.

This method is used often in analyses.

**Signature:**

```
<Column> = [Table].column(<str> col)
```

- **<str> col:** The column name to return

Returns: A Column object representing the specified column of the table.

### 7.139 `Table.column_count`

Retrieves the number of columns in a table. Note that since Picalo lists are zero-based, you access individual columns starting with 0.

So although `column_count()` may report 3 columns, you access them via `table.column(0)`, `table.column(1)`, and `table.column(2)`.

However, using column names rather than direct indices is easier and more readable. `table['colname']` returns the given column.

**Signature:**

```
<int> = [Table].column_count()
```

Returns: The number columns in the table

## 7.140 Table.delete\_column

Removes a column from the table and discards the records. Remaining column indices are decremented to reflect the new table structure. Column names (columns) are not modified.

This action is permanent and cannot be undone.

### Signature:

```
[Table].delete_column(<str or list> column)
```

- **<str or list> column:** The name or index of the column to be removed. If a list of columns, all of the columns are removed.

## 7.141 Table.deref\_column

Dereferences the col name to its index (if it is a name). For example, if the column name "id" is given, it returns the index of this column (such as 0, 1, 2, etc.).

The column can be specified as the column index, the column name, or even a negative index (from the last column backward).

### Signature:

```
<int> = [Table].deref_column(<str/int> col_name)
```

- **<str/int> col\_name:** The name of the column

Returns: The index of the column



## 7.142 Table.extend

Appends the records in the given table to the end of this table. The two tables must have the same number of columns to be merged.

### Signature:

```
[Table].extend(<Table> table)
```

- **<Table> table:** The records of the specified table will be added to this table.

### Example:

```
1 >>> mytable3.extend(mytable2) # appends mytable2 to the end of mytable3
```

## 7.143 Table.filter

Filters the table with the given expression. Until the filter is either replaced or cleared, only the records matching the filter will be available in the table. All Picalo functions will see only this limited view of the table in their analyses. In other words, it is as if the filtered record is not in the table at all until the filter is removed.

Filters are transient. They do not save with the table and they do not carry over if you copy a table. They are simply temporary filters that you can use to restrict Picalo analyses to a few records.

Only one filter can be active at any time. Setting a new filter will replace the existing one.

In creating your expression, use the standard record['col'] notation to access individual records in the table.

### Signature:

```
[Table].filter(<str> expression="None")
```

- **<str> expression (optional):** A valid Picalo expression that returns True or False. If the If omitted, defaults to None.

## 7.144 Table.find

Finds the record indices with given key=record pairs. This method is not normally used directly – use `Simple.select_by_record` instead.

This method is different than `Simple.select_by_record` in that it does not create a new table. `Simple.select_by_record` creates a new table consisting of copies of all matching records. In contrast, this method simply returns the record indices of the matching records.

This method *is* efficient and can be used often. It calculates indices as needed and should select very fast.

### Signature:

```
<list> = [Table].find()
```

Returns: A list of indices that match the pairs.

### Example:

```
1 >>> table = Table([
2 ...   ('col001', int),
3 ...   ('col002', int),
4 ...   ('col003', unicode),
5 ... ], [
6 ...   [5,6,'flo'],
7 ...   [3,2,'sally'],
8 ...   [4,6,'dan'],
9 ...   [4,7,'stu'],
10 ...  [4,7,'ben'],
11 ...  [4,6,'benny'],
12 ... ])
13 >>> results = table.find(col001=6, col000=4)
14 >>> results
15 [2, 5]
```

## 7.145 Table.get\_column\_names

Returns the column names of this table.

### Signature:

```
<list> = [Table].get_column_names()
```

Returns: A list of string names of columns

## 7.146 `Table.get_columns`

Returns the column objects of this table. Columns are `Column` objects, which contain the column name, calculated expression (if any), type if set, etc.

Most users should call `get_column_names()` instead as it only returns the names of the columns. Only call this method if you need to access the internal column objects.

### Signature:

```
<list> = [Table].get_columns()
```

Returns: A list of `Column` objects

## 7.147 `Table.get_filter_expression`

Returns the filter expression as a `PicaloExpression` object, or `None` if no filter is applied.

### Signature:

```
[Table].get_filter_expression()
```

## 7.148 `Table.guess_types`

Inspects the first `num_records` in each column and automatically sets the type of each column. This method is not perfect, but it is fairly robust. It is a good method to call after importing from TSV/CSV if you don't want to set types manually. If `num_records` is less than 1, all records in the table are inspected (this can take a long time for huge tables but should be done for all other tables).

If a column is a calculated column, its type is not set (columns with expressions do not have an explicit type).

### Signature:

```
[Table].guess_types(<int> num_records=-1)
```

- `<int> num_records` (optional): The number of records to inspect before setting the type. If omitted, defaults to -1.

## 7.149 Table.index

Returns an index on this table for the given column name(s). This method allows you to find the record indices that match specific keys. If only one column name is specified, the index will have as many records as there are unique records in the column. If multiple columns are specified, the index will have as many records as there are unique combinations of the records in the columns.

Indices are used throughout Picalo internally and are not normally accessed by users. However, many analyses need to calculate indices directly, and exposing this method allows this behavior.

This method is efficient – if the table data have not been modified since the last time a specific index was asked for, it uses the previously calculated index.

### Signature:

```
[Table].index()
```

## 7.150 Table.insert

Inserts a new record at the given index location. The first parameter *\*must\** be the index location to insert the record. The remaining parameters are the same as the `append()` method. Possible formats are (assuming you want to place the new record in the second row and push all existing records down one):

```
Format 1: - newrec = mytable3.insert(2) # insert before row 2 - newrec['ID']
= 4 - newrec['Name'] = 'Homer' - newrec['Salary'] = 15000
```

```
Format 2: - mytable.insert(2, 4, 'Homer', 1500) # insert before row 2
```

```
Format 3: - mytable3.insert(2, [4, 'Homer', 1500]) # insert before row 2
```

```
Format 4: - mytable3.insert(2, {'ID':5, 'Name':'Marge', 'Salary': 275000})
# insert before row 2
```

```
Format 5: - mytable3.insert(2, {0:5, 1:'Marge', 2: 275000}) # insert
before row 2
```

Format 6: - mytable3.insert(2, ID=5, Name='Krusty', Salary=50000) #  
insert before row 2

You cannot mix formats in the same call.

**Signature:**

```
<returns> = [Table].insert()
```

Returns: The new record object (that was inserted to the end of the table)

## 7.151 Table.insert\_calculated

Inserts a new, calculated column with records given by expression at the given index location. Calculated columns act as regular columns in all ways. Their records are 'active' meaning their records change when the result of the expression. In other words, they are recalculated each time they are used rather than being stored statically. This is similar to the way Excel functions always reflect the most updated data records.

**Signature:**

```
<Column> = [Table].insert_calculated(<int> index,  
                                     <str> name,  
                                     <str> expression)
```

- **<int> index:** The index location of the new column. Previous column indices are incremented one to make room for the new column.
- **<str> name:** The new column name
- **<str> expression:** An expression that returns the record of the new field. As shown in the example, use rec to denote the current record being evaluated.

Returns: The new column object.

**Example:**

```

1 >>> table = Table([('id', int)], [[1],[2],[4]])
2 >>> table.insert_calculated(0, 'plusone', "col000+1")
3 >>> table.view()
4 +-----+
5 | plusone | col000 |
6 +-----+
7 |        2 |        1 |
8 |        3 |        2 |
9 |        5 |        4 |
10 +-----+

```

## 7.152 Table.insert\_calculated\_static

Inserts a new, calculated column with records given by expression at the given index location. The records are calculated immediately using expression, and then they are static. In other words, this method calculates a new, regular column. The records are not 'active' in the sense that `append_calculated()` columns are active. When this method returns, the new column is the same as any other, non-calculated column.

### Signature:

```

<Column> = [Table].insert_calculated_static(<int> index,
                                             <str> name,
                                             <type> column_type,
                                             <str> expression)

```

- **<int> index:** The index location of the new column. Previous column indices are incremented one to make room for the new column.
- **<str> name:** The new column name
- **<type> column\_type:** The type of the new column (str, Date, int, long, etc.)
- **<str> expression:** An expression that returns the record of the new field. As shown in the example, use `rec` to denote the current record being evaluated.

Returns: The new column object.

### Example:

```

1 >>> table = Table([('id', int)], [[1],[2],[4]])
2 >>> table.append_calculated('plusone', int, "col000 + 1")
3 >>> table.view()
4 +-----+
5 | col000 | plusone |
6 +-----+
7 |      1 |       2 |
8 |      2 |       3 |
9 |      4 |       5 |
10 +-----+

```

### 7.153 Table.insert\_column

Inserts a new column in the table at the given index location.

#### Signature:

```

<Column> = [Table].insert_column(<object> index,
                                <str> name,
                                <type> column_type,
                                <list,
                                Column,
                                or Table> records=None)

```

- **<object> index:**
- **<str> name:** The new column name
- **<type> column\_type:** The new column type (int, float, DateTime, unicode, str, etc)
- **<list, Column, or Table> records (optional):** A list of records to place into the cells of the new column (if a full Table, the first column is used) If omitted, defaults to None.

Returns: The new column object.

### 7.154 Table.is\_changed

Returns whether the table has been changed since loading

#### Signature:

`[Table].is_changed()`

### 7.155 `Table.is_filtered`

Returns True if this table has an active filter

**Signature:**

`[Table].is_filtered()`

### 7.156 `Table.is_readonly`

Returns whether this table is read only.

**Signature:**

`<bool> = [Table].is_readonly()`

Returns: Whether this table is read only.

### 7.157 `Table.iterator`

Returns an iterator to the Records in thisTable. This is normally achieved through: `>>> for record in mytable: >>> # do something with each record object`

This method is provided to allow you to ignore the filter if you want. The regular iterator syntax (`for record in mytable`) always respects any active filters.

**Signature:**

`<iterator> = [Table].iterator(<object> respect_filter=True)`

- `<object> respect_filter` (optional): If omitted, defaults to True.

Returns: An iterator to this Table.



## 7.158 Table.move\_column

Moves a column to another location in the table. A column can be moved in front of other columns or behind other columns with this method.

The column parameter is the name of the column to be moved, the index of the column to be moved, or the column object itself.

The new\_index parameter is the new index for this column. This can be seen as the insertion point, or the column the moved column will be placed in front of. It can be specified as a column index, a column name, or a column object.

### Signature:

```
[Table].move_column(<int/str/Column> column,  
                   <int/str/Column> new_index)
```

- **<int/str/Column> column:** The name, index, or column object to be moved.
- **<int/str/Column> new\_index:** The name, index, or column object that the column will be placed before.

## 7.159 Table.prettyprint

Pretty prints the table to the given fp. Note that the preferred way to print a table is to call "table.view()", which opens the table in the Picalo GUI if possible, or uses view() if in console mode. In other words, you should normally use view() rather than this method.

### Signature:

```
[Table].prettyprint(<file> fp=None,  
                   <bool> center_columns=True,  
                   <str> space_before=" ",  
                   <str> space_after=" ",  
                   <str> col_separator_char="|",  
                   <str> row_separator_char="-",  
                   <str> join_char="+",  
                   <str> line_ending="")
```

",

```
<str> none="<N>",
<str> encoding="utf-8",
<object> respect_filter=True)
```

- `<file> fp` (optional): An open file pointer object. If `None`, defaults to standard output stream. If omitted, defaults to `None`.
- `<bool> center_columns` (optional): Whether to center columns or not. If omitted, defaults to `True`.
- `<str> space_before` (optional): An optional spacing between the leading column separator and the field record. If omitted, defaults to `.`
- `<str> space_after` (optional): An optional spacing between the field and the trailing column separator. If omitted, defaults to `.`
- `<str> col_separator_char` (optional): An optional column separator character to use in the printout. If omitted, defaults to `—`.
- `<str> row_separator_char` (optional): An optional row separator character to use in the printout. If omitted, defaults to `-`.
- `<str> join_char` (optional): An optional character to use when joining rows and columns. If omitted, defaults to `+`.
- `<str> line_ending` (optional): An optional line ending character(s) to use. If omitted, defaults to `.`
- `<str> none` (optional): The record to print when cells are set to the special `None` record. If omitted, defaults to `<N>`.
- `<str> encoding` (optional): The unicode encoding to write with. This should be a value from the `codecs` module. If `None`, the encoding is guessed to `utf-8`, `utf-16`, `utf-16-be`, or `utf-16-le`. If omitted, defaults to `utf-8`.
- `<object> respect_filter` (optional): If omitted, defaults to `True`.

## 7.160 Table.record

Retrieves the record at the given index. This is one of the most-used methods in the Picalo toolkit as it gives you access to records.

In keeping with most computer languages, Picalo indices are always zero-based. This may require a slight adjustment for some users, but it makes mathematical calculations much easier and has other implications. This means that record 1 is `table[0]`, record 2 is `table[1]`, and so forth.

Note that the shortcut way to access this method is the simple `[n]` notation, as in `table[1]` to access the second record. `table.record()` is rarely called as the shortcut is preferred instead.

For advanced users: Index can also be a slice, as in `table[2:5]` to return a new Picalo table including only records 2, 3, and 4. See the Python documentation for more information on slices.

The method respects any filters on the table by default. This can be overridden to ignore the filter.

### Signature:

```
<Record> = [Table].record(<int> index,
                          <object> respect_filter=True)
```

- **<int> index:** The zero-based index of the record to pull.
- **<object> respect\_filter (optional):** If omitted, defaults to True.

Returns: A Picalo Record object, which allows access to members via column name.

### Example 1:

```
1 >>> table = Table([('id', int)], [[1],[2],[3],[4]])
2 >>> table2 = table[1]
3 >>> # table2 is now a Record object pointing at [2]
```

### Example 2:

```
1 >>> table = Table([('id', int)], [[1],[2],[3],[4]])
2 >>> print table[1]['col000']
3 2
```

## 7.161 Table.record\_count

Returns the number of records in this table. The method respects any active filters on the table by default.

**Signature:**

```
<int> = [Table].record_count(<object> respect_filter=True)
```

- **<object> respect\_filter** (optional): If omitted, defaults to True.

Returns: The number of records in the table.

## 7.162 Table.reorder\_columns

Reorders the columns according to the given list. This is an alternative to `move_column`. If you know the exact order you want the columns in, use this method to explicitly set them.

The `columns` parameter is a list giving the new order. Its items can be current column indices, names, or column objects.

**Signature:**

```
[Table].reorder_columns(<list> columns)
```

- **<list> columns**: A list giving the new column order (use column names, not indices).

## 7.163 Table.save

Saves this table in native Picalo format. This is the preferred format to save tables in because all column types, formulas, and so forth are saved.

**Signature:**

```
[Table].save(<str> filename,  
             <object> respect_filter=False)
```

- **<str> filename**: The filename to save to. This can also be an open stream.
- **<object> respect\_filter** (optional): If omitted, defaults to False.

## 7.164 Table.save\_csv

Saves this table to a Comma Separated Values (CSV) text file. CSV is an industry-standard way of transferring data between applications. This is the preferred way of exporting data from Picalo.

Note that although this is the preferred export method, it has some limitations. These are limitations of the format rather than limitations of Picalo:

- No type information is saved to the file. All data is essentially turned into strings.
- Be sure to use the correct encoding if using international languages.
- Different standards for CSV exist (that's the nice thing about standards :).

This export uses the Microsoft version.

Note that Microsoft Office seems to like CSV files better than TSV files.

### Signature:

```
[Table].save_csv(<str> filename,  
                <str> line_ending="  
",  
                <str> none="",  
                <str> encoding="utf-8",  
                <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with. If omitted, defaults to `.`
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to `.`
- **<str> encoding (optional):** The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.165 Table.save\_delimited

Saves this table to a delimited text file. This method allows the specification of different types of delimiters and qualifiers.

This method is for advanced users. Most users should call `save_tsv`, `save_csv`, or `save_fixed` to save using one of the accepted text formats. See these methods for more information.

### Signature:

```
[Table].save_delimited(<str> filename,
                      <str> delimiter=",",
                      <str> qualifier="\"",
                      <str> line_ending="
",
                      <str> none="",
                      <str> encoding="utf-8",
                      <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> delimiter (optional):** A field delimiter character, defaults to a comma (,) If omitted, defaults to ,.
- **<str> qualifier (optional):** A qualifier to use when delimiters exist in field records, defaults to a double quote (") If omitted, defaults to ".
- **<str> line\_ending (optional):** A line ending to separate rows with, defaults to `os.linesep` ( If omitted, defaults to .
- **<str> none (optional):** An parameter specifying what to write for cells that have the `None` value, defaults to an empty string (") If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to write with. This should be a value from the `codecs` module, defaults to 'utf-8'. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to `False`.

## 7.166 Table.save\_excel

Saves this table to a Microsoft Excel 97+ file.

**Signature:**

```
[Table].save_excel(<str> filename,  
                  <str> none="",  
                  <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to .
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.167 Table.save\_fixed

Saves this table to a fixed width text file. Fixed width files pad column records with extra spaces so they are easier to read.

You should normally prefer CSV and TSV export formats to fixed as they have less limitations. Fixed format was widely used in the early days of computers, and some servers still use the format.

**Signature:**

```
[Table].save_fixed(<str> filename,  
                  <str> line_ending="  
",  
                  <str> none="",  
                  <object> encoding=utf-8,  
                  <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.

- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to `.`
- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` record. If omitted, defaults to `.`
- `<object> encoding` (optional): If omitted, defaults to `utf-8`.
- `<object> respect_filter` (optional): If omitted, defaults to `False`.

## 7.168 `Table.save_tsv`

Saves this table to a Tab Separated Values (TSV) text file. TSV is an industry-standard way of transferring data between applications.

Note that although this is the preferred export method, it has some limitations. These are limitations of the format rather than limitations of Picalo:

- No type information is saved to the file. All data is essentially turned into strings.
- Be sure to use the correct encoding if using international languages.
- Different standards for TSV exist (that's the nice thing about standards :).

This export uses the Microsoft version.

Note that Microsoft Office seems to like CSV files better than TSV files.

### Signature:

```
[Table].save_tsv(<str> filename,
                 <str> line_ending="
",
                 <str> none="",
                 <str> encoding="utf-8",
                 <object> respect_filter=False)
```

- **`<str> filename`:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to `.`
- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` record. If omitted, defaults to `.`



- `<str> encoding` (optional): The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- `<object> respect_filter` (optional): If omitted, defaults to False.

## 7.169 Table.save\_xml

Saves this table to an XML file using a pre-defined schema. If you need to save to a different schema, use the `xml.dom.minidom` class directly.

**Signature:**

```
[Table].save_xml(<str> filename,
                 <str> line_ending="
",
                 <str> indent=" ",
                 <bool> compact=False,
                 <str> none="",
                 <object> encoding=utf-8,
                 <object> respect_filter=False)
```

- **`<str> filename`:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> line_ending` (optional): An optional line ending to separate rows with (when `compact` is False) If omitted, defaults to `.`
- `<str> indent` (optional): The character(s) to use for indenting (when `compact` is False) If omitted, defaults to `.`
- `<bool> compact` (optional): Whether to compact the XML or make it "pretty" with whitespace If omitted, defaults to False.
- `<str> none` (optional): An optional parameter specifying what to write for cells that have the None record. If omitted, defaults to `.`

- `<object>` encoding (optional): If omitted, defaults to utf-8.
- `<object>` respect\_filter (optional): If omitted, defaults to False.

## 7.170 `Table.set_changed`

Sets whether the class has been changed since loading. This is not normally called by users.

### Signature:

```
[Table].set_changed(<object> changed)
```

- `<object>` changed:

## 7.171 `Table.set_format`

Sets the format of this column. The format is used for printing the record and showing the record in the Picalo GUI.

The format should be a Picalo expression that evaluates to a string. Use the 'record' variable for the record of the current cell.

Note that this is not an input mask. It doesn't affect the internal record of the field records. It only affects how it is displayed on the screen.

### Signature:

```
[Table].set_format(<str> column,
                  <str> format="None")
```

- `<str>` column: The column name or index to set the type of
- `<str>` format (optional): A Picalo expression that evaluates to a string. If omitted, defaults to None.

### Example:

```
1 # shows the current record in uppercase
2 table.set_format('Salary', "record.upper()")
```

## 7.172 Table.set\_name

Changes the name of an existing column. The column name must be a valid Picalo name and must be unique to other column names in the table.

**Signature:**

```
[Table].set_name(<object> column,  
                <object> name)
```

- **<object> column:**
- **<object> name:**

## 7.173 Table.set\_readonly

Sets the read only status of this table. Tables that are read only cannot be modified. Normally, tables are initially not read only (i.e. can be modified). The only exception is tables loaded from databases, which are read only.

**Signature:**

```
[Table].set_readonly(<bool> readonly_flag=False)
```

- **<bool> readonly\_flag (optional):** True or False, depending upon whether the table should be read only or not. If omitted, defaults to False.

## 7.174 Table.set\_type

Sets the type of a column. The type must be a valid <type> object, such as int, float, str, unicode, DateTime, etc. All records in this column will be converted to this new type.

The format is an optional format to be used for printing the record and showing the record in the Picalo GUI.

**Signature:**

```
[Table].set_type(<str> column,
                <type> column_type=None,
                <str> format="None",
                <None> expression=None)
```

- **<str> column:** The column name or index to set the type of
- **<type> column\_type (optional):** The new type of this column. If omitted, defaults to None.
- **<str> format (optional):** A Picalo expression that evaluates to a string. If omitted, defaults to None.
- **<None> expression (optional):** A Picalo expression that calculates this column. If omitted, defaults to None.

## 7.175 Table.sort

Sorts this table with optional arguments.

**Signature:**

```
[Table].sort(<function> cmp=None,
             <function> key=None,
             <bool> reverse=False)
```

- **<function> cmp (optional):** An optional function that compares two items. If omitted, defaults to None.
- **<function> key (optional):** A function that takes a single item and returns a version of it for use in sorting. If omitted, defaults to None.
- **<bool> reverse (optional):** Whether to sort in reverse. If omitted, defaults to False.

## 7.176 Table.structure

Returns the structure of this table, including column names, input and output types, and general statistics.

**Signature:**

```
<Table> = [Table].structure()
```

Returns: A Picalo table describing the structure of this table.

### 7.177 Table.view

Opens a spreadsheet-view of the table if Picalo is being run in GUI mode. If Picalo is being run in console mode, it redirects to `prettyprint()`. This is the preferred way of viewing the data in a table.

**Signature:**

```
[Table].view()
```

### 7.178 TableArray

A list of picalo tables. Some functions in Picalo return a set of tables. Sets of tables are returned as `TableArray` objects. These objects have the exact same methods and behavior as typical Python lists.

For example, the `Grouping.stratify_by_value()` method stratifies a table into a number of smaller tables. These tables are returned in `TableArray` objects. You can access individual tables in a `TableArray` using the `[n]` notation.

Users do not normally create `TableArrays`. They are created automatically by Picalo functions.

`TableArrays` must hold the exact same type of table. For example, two tables in a given `TableArray` may not have different column names. If you need to hold a list of noncompatible tables (i.e. tables with different columns or types), use a regular python list `[]`.

**Signature:**

```
TableArray()
```

**Example:**

```

1 >>> data = Table([
2 ...     ('id', int),
3 ...     ('name', unicode),
4 ... ],[
5 ...     [ 1, 'Benny' ],
6 ...     [ 2, 'Vijay' ],
7 ... ])
8 >>> tables = Grouping.stratify_by_value(data, 'id')
9 >>> tables[0].view()
10 +-----+
11 | id | name |
12 +-----+
13 |  1 | Benny |
14 +-----+
15 >>> tables[1].view()
16 +-----+
17 | id | name |
18 +-----+
19 |  2 | Vijay |
20 +-----+
21 >>> for table in tables:
22 ...     print len(table)
23 ...
24 1
25 1

```

## 7.179 TableArray.append

None

### Signature:

[TableArray].append(<object> value)

- <object> value:

## 7.180 TableArray.append\_calculated

Appends a calculated column to each table in the list. See the definition of this method in Table for more information.

### Signature:

[TableArray].append\_calculated(<object> name,  
                                   <object> expression)

- **<object> name:**
- **<object> expression:**

## 7.181 TableArray.append\_calculated\_static

Appends a calculated column to each table in the list. See the definition of this method in Table for more information.

**Signature:**

```
[TableArray].append_calculated_static(<object> name,  
                                       <object> column_type,  
                                       <object> expression)
```

- **<object> name:**
- **<object> column\_type:**
- **<object> expression:**

## 7.182 TableArray.append\_column

Adds a new column to each table in the list. See the definition of this method in Table for more information.

**Signature:**

```
[TableArray].append_column(<object> name,  
                           <object> column_type,  
                           <object> values=None)
```

- **<object> name:**
- **<object> column\_type:**
- **<object> values (optional):** If omitted, defaults to None.

### 7.183 `TableArray.clear_filter`

Clears any active filter to all tables in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].clear_filter()
```

### 7.184 `TableArray.column`

Returns the given column from the first table in this list. Since all tables have the same column definitions, it is comparable across all tables in the list.

**Signature:**

```
[TableArray].column(<object> col)
```

- `<object> col`:

### 7.185 `TableArray.column_count`

Returns the column count from the first table in this list. Since all tables have the same column definitions, it is comparable across all tables in the list.

**Signature:**

```
[TableArray].column_count()
```

### 7.186 `TableArray.combine`

Combines this table array into a single table. This is a kind of 'anti-stratification'.

**Signature:**

```
[TableArray].combine()
```



## 7.187 `TableArray.delete_column`

Deletes a column from each table in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].delete_column(<object> column)
```

- `<object> column`:

## 7.188 `TableArray.filter`

Applies the given filter to all tables in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].filter(<object> expression=None)
```

- `<object> expression` (optional): If omitted, defaults to `None`.

## 7.189 `TableArray.get_column_names`

Returns the column names from the first table in this list. Since all tables have the same column definitions, it is comparable across all tables in the list.

**Signature:**

```
[TableArray].get_column_names()
```

## 7.190 `TableArray.get_columns`

Returns the columns from the first table in this list. Since all tables have the same column definitions, it is comparable across all tables in the list.

**Signature:**

```
[TableArray].get_columns()
```

### 7.191 `TableArray.get_filter_expression`

Returns the filter expression as a `PicaloExpression` object, or `None` if no filter is applied.

**Signature:**

```
[TableArray].get_filter_expression()
```

### 7.192 `TableArray.guess_types`

Guesses the column types for all tables in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].guess_types(<object> num_records=-1)
```

- `<object> num_records` (optional): If omitted, defaults to -1.

### 7.193 `TableArray.insert_calculated`

Inserts a calculated column to each table in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].insert_calculated(<object> index,  
                               <object> name,  
                               <object> expression)
```

- `<object> index`:
- `<object> name`:
- `<object> expression`:

## 7.194 `TableArray.insert_calculated_static`

Inserts a calculated column to each table in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].insert_calculated_static(<object> index,  
                                     <object> name,  
                                     <object> column_type,  
                                     <object> expression)
```

- `<object> index`:
- `<object> name`:
- `<object> column_type`:
- `<object> expression`:

## 7.195 `TableArray.insert_column`

Inserts a new column to each table in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].insert_column(<object> index,  
                           <object> name,  
                           <object> column_type,  
                           <object> values=None)
```

- `<object> index`:
- `<object> name`:
- `<object> column_type`:
- `<object> values` (optional): If omitted, defaults to `None`.

## 7.196 `TableArray.is_changed`

Returns whether the table has been changed since loading

**Signature:**

```
[TableArray].is_changed()
```

## 7.197 `TableArray.is_filtered`

Returns whether the first table in this list is filtered.

**Signature:**

```
[TableArray].is_filtered()
```

## 7.198 `TableArray.is_readonly`

Returns whether this table is read only.

**Signature:**

```
<bool> = [TableArray].is_readonly()
```

Returns: Whether this table is read only.

## 7.199 `TableArray.move_column`

Moves a column to another location for each table in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].move_column(<object> column,  
                        <object> new_index)
```

- **<object> column:**
- **<object> new\_index:**

## 7.200 `TableArray.save`

Saves this `TableList` in native Picalo format. This is the preferred format to save `TableLists` in because all column types, formulas, and so forth are saved.

**Signature:**

```
[TableArray].save(<str> filename,
                  <object> respect_filter=False)
```

- **<str> filename:** The filename to save to. This can also be an open stream.
- **<object> respect\_filter (optional):** If omitted, defaults to `False`.

## 7.201 `TableArray.save_csv`

Saves this `TableList` in CSV format. Since the table list probably has multiple tables in it, one CSV per table is created by prepending 1, 2, 3 to the filename.

**Signature:**

```
[TableArray].save_csv(<str> filename,
                      <str> line_ending="
",
                      <str> none="",
                      <str> encoding="utf-8",
                      <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with. If omitted, defaults to `.`
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the `None` value. If omitted, defaults to `.`

- `<str> encoding` (optional): The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- `<object> respect_filter` (optional): If omitted, defaults to False.

## 7.202 TableArray.save\_delimited

Saves this TableList in delimited format. Since the table list probably has multiple tables in it, one delimited file per table is created by prepending 1, 2, 3 to the filename.

### Signature:

```
[TableArray].save_delimited(<str> filename,
                             <str> delimiter=",",
                             ",
                             <str> qualifier="",
                             <str> line_ending="
",
                             <str> none="",
                             <str> encoding="utf-8",
                             <object> respect_filter=False)
```

- **`<str> filename`:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> delimiter` (optional): An optional field delimiter character. If omitted, defaults to ,.
- `<str> qualifier` (optional): An optional qualifier to use when delimiters exist in field values. If omitted, defaults to " .
- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to .

- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` value. If omitted, defaults to `.`
- `<str> encoding` (optional): The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to `utf-8`.
- `<object> respect_filter` (optional): If omitted, defaults to `False`.

### 7.203 `TableArray.save_fixed`

Saves this `TableList` in fixed format. Since the table list probably has multiple tables in it, one fixed file per table is created by prepending 1, 2, 3 to the filename.

#### Signature:

```
[TableArray].save_fixed(<str> filename,
                        <str> line_ending="
",
                        <str> none="",
                        <object> respect_filter=False)
```

- **`<str> filename`:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to `.`
- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` value. If omitted, defaults to `.`
- `<object> respect_filter` (optional): If omitted, defaults to `False`.

## 7.204 `TableArray.save_tsv`

Saves this TableList in tsv format. Since the table list probably has multiple tables in it, one tsv file per table is created by prepending 1, 2, 3 to the filename.

**Signature:**

```
[TableArray].save_tsv(<str> filename,
                      <str> line_ending="
",
                      <str> none="",
                      <str> encoding="utf-8",
                      <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with If omitted, defaults to .
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None value. If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.205 `TableArray.save_xml`

Saves this TableList in xml format. Since the table list probably has multiple tables in it, one xml file per table is created by prepending 1, 2, 3 to the filename.

**Signature:**



```
[TableArray].save_xml(<str> filename,
                      <str> line_ending="
",
                      <str> indent=" ",
                      <bool> compact=False,
                      <str> none="",
                      <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with (when compact is False) If omitted, defaults to .
- **<str> indent (optional):** The character(s) to use for indenting (when compact is False) If omitted, defaults to .
- **<bool> compact (optional):** Whether to compact the XML or make it "pretty" with whitespace If omitted, defaults to False.
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None value. If omitted, defaults to .
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.206 TableArray.set\_changed

Sets whether the class has been changed since loading. This is not normally called by users.

**Signature:**

```
[TableArray].set_changed(<object> changed)
```

- **<object> changed:**

## 7.207 `TableArray.set_format`

Sets the format for a column in all tables in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].set_format(<object> column,  
                        <object> format=None)
```

- **<object> column:**
- **<object> format (optional):** If omitted, defaults to `None`.

## 7.208 `TableArray.set_name`

Sets the column name for all tables in the list. See the definition of this method in `Table` for more information.

**Signature:**

```
[TableArray].set_name(<object> column,  
                     <object> name)
```

- **<object> column:**
- **<object> name:**

## 7.209 `TableArray.set_readonly`

Sets the read only status of this table. Tables that are read only cannot be modified. Normally, tables are initially not read only (i.e. can be modified). The only exception is tables loaded from databases, which are read only.

**Signature:**

```
[TableArray].set_readonly(<bool> readonly_flag=False)
```

- **<bool> readonly\_flag (optional):** True or False, depending upon whether the table should be read only or not. If omitted, defaults to `False`.

## 7.210 `TableArray.set_type`

Sets the column type for all tables in the list. See the definition of this method in `Table` for more information.

### Signature:

```
[TableArray].set_type(<object> column,  
                     <object> column_type=None,  
                     <object> format=None,  
                     <object> expression=None)
```

- **<object> column:**
- **<object> column\_type** (optional): If omitted, defaults to `None`.
- **<object> format** (optional): If omitted, defaults to `None`.
- **<object> expression** (optional): If omitted, defaults to `None`.

## 7.211 `TableArray.structure`

Returns the structure of the first table in this list. Since all tables have the same column definitions, it is comparable across all tables in the list.

### Signature:

```
[TableArray].structure()
```

## 7.212 `TableArray.view`

Opens the table list for viewing in the Picalo user interface. The resulting view allows you to page through the tables in the list. See the first example.

You can view individual tables in the list by using the `[n]` notation. See the second example for this notation.

### Signature:

```
[TableArray].view()
```

**Example 1:**

```

1 >>> data = Table([
2 ...     ('id', int),
3 ...     ('name', unicode),
4 ... ], [
5 ...     [ 1, 'Benny' ],
6 ...     [ 2, 'Vijay' ],
7 ... ])
8 >>> tables = Grouping.stratify_by_value(data, 'id')
9 >>> tables.view()

```

**Example 2:**

```

1 >>> data = Table([
2 ...     ('id', int),
3 ...     ('name', unicode),
4 ... ], [
5 ...     [ 1, 'Benny' ],
6 ...     [ 2, 'Vijay' ],
7 ... ])
8 >>> tables = Grouping.stratify_by_value(data, 'id')
9 >>> tables[0].view()

```

## 7.213 TableList

A list of picalo tables. Some functions in Picalo return a set of tables. Sets of tables are returned as TableList objects.

Users do not normally create TableLists. They are created automatically by Picalo functions.

TableLists are a weaker form of TableArrays. While TableArrays must have compatible tables, TableLists can hold tables of different schema. TableLists can be viewed in the Picalo GUI, but they cannot be sent into most Picalo functions.

**Signature:**

TableList()

## 7.214 TableList.append

None

**Signature:**

```
[TableList].append(<object> value)
```

- <object> value:

## 7.215 TableList.append\_calculated

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].append_calculated(<object> name,  
                              <object> expression)
```

- <object> name:
- <object> expression:

## 7.216 TableList.append\_calculated\_static

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].append_calculated_static(<object> name,  
                                     <object> column_type,  
                                     <object> expression)
```

- <object> name:
- <object> column\_type:
- <object> expression:

## 7.217 `TableList.append_column`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].append_column(<object> name,  
                           <object> column_type,  
                           <object> values=None)
```

- **<object> name:**
- **<object> column\_type:**
- **<object> values (optional):** If omitted, defaults to None.

## 7.218 `TableList.clear_filter`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].clear_filter()
```

## 7.219 `TableList.column`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].column(<object> col)
```

- **<object> col:**

## 7.220 `TableList.column_count`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].column_count()
```

## 7.221 `TableList.delete_column`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].delete_column(<object> column)
```

- `<object> column`:

## 7.222 `TableList.filter`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].filter(<object> expression=None)
```

- `<object> expression` (optional): If omitted, defaults to None.

## 7.223 `TableList.get_column_names`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].get_column_names()
```

## 7.224 `TableList.get_columns`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].get_columns()
```

## 7.225 `TableList.guess_types`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].guess_types(<object> num_records=-1)
```

- `<object> num_records` (optional): If omitted, defaults to -1.

## 7.226 `TableList.insert_calculated`

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].insert_calculated(<object> index,  
                              <object> name,  
                              <object> expression)
```

- `<object> index`:
- `<object> name`:
- `<object> expression`:



## 7.227 `TableList.insert_calculated_static`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].insert_calculated_static(<object> index,  
                                     <object> name,  
                                     <object> column_type,  
                                     <object> expression)
```

- `<object> index`:
- `<object> name`:
- `<object> column_type`:
- `<object> expression`:

## 7.228 `TableList.insert_column`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].insert_column(<object> index,  
                           <object> name,  
                           <object> column_type,  
                           <object> values=None)
```

- `<object> index`:
- `<object> name`:
- `<object> column_type`:
- `<object> values` (optional): If omitted, defaults to None.

## 7.229 TableList.is\_changed

Returns whether the table has been changed since loading

**Signature:**

```
[TableList].is_changed()
```

## 7.230 TableList.is\_filtered

Returns whether the first table in this list is filtered.

**Signature:**

```
[TableList].is_filtered()
```

## 7.231 TableList.is\_readonly

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].is_readonly()
```

## 7.232 TableList.move\_column

This method is not supported in TableLists. Use TableArrays if this method is required.

**Signature:**

```
[TableList].move_column(<object> column,  
                        <object> new_index)
```

- **<object> column:**
- **<object> new\_index:**

## 7.233 TableList.save

Saves this TableList in native Picalo format. This is the preferred format to save TableLists in because all column types, formulas, and so forth are saved.

### Signature:

```
[TableList].save(<str> filename,
                 <object> respect_filter=False)
```

- **<str> filename:** The filename to save to. This can also be an open stream.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.234 TableList.save\_csv

Saves this TableList in CSV format. Since the table list probably has multiple tables in it, one CSV per table is created by prepending 1, 2, 3 to the filename.

### Signature:

```
[TableList].save_csv(<str> filename,
                    <str> line_ending="
",
                    <str> none="",
                    <str> encoding="utf-8",
                    <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with If omitted, defaults to .
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None value. If omitted, defaults to .

- `<str> encoding` (optional): The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- `<object> respect_filter` (optional): If omitted, defaults to False.

## 7.235 TableList.save\_delimited

Saves this TableList in delimited format. Since the table list probably has multiple tables in it, one delimited file per table is created by prepending 1, 2, 3 to the filename.

**Signature:**

```
[TableList].save_delimited(<str> filename,
                           <str> delimiter=",",
                           <str> qualifier="\"",
                           <str> line_ending="
",
                           <str> none="",
                           <str> encoding="utf-8",
                           <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> delimiter` (optional): An optional field delimiter character. If omitted, defaults to `,`.
- `<str> qualifier` (optional): An optional qualifier to use when delimiters exist in field values. If omitted, defaults to `"`.
- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to `.`

- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` value. If omitted, defaults to `.`
- `<str> encoding` (optional): The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to `utf-8`.
- `<object> respect_filter` (optional): If omitted, defaults to `False`.

### 7.236 `TableList.save_fixed`

Saves this `TableList` in fixed format. Since the table list probably has multiple tables in it, one fixed file per table is created by prepending 1, 2, 3 to the filename.

#### Signature:

```
[TableList].save_fixed(<str> filename,
                      <str> line_ending="
",
                      <str> none="",
                      <object> respect_filter=False)
```

- **`<str> filename`:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- `<str> line_ending` (optional): An optional line ending to separate rows with. If omitted, defaults to `.`
- `<str> none` (optional): An optional parameter specifying what to write for cells that have the `None` value. If omitted, defaults to `.`
- `<object> respect_filter` (optional): If omitted, defaults to `False`.

## 7.237 TableList.save\_tsv

Saves this TableList in tsv format. Since the table list probably has multiple tables in it, one tsv file per table is created by prepending 1, 2, 3 to the filename.

### Signature:

```
[TableList].save_tsv(<str> filename,  
                    <str> line_ending="  
",  
                    <str> none="",  
                    <str> encoding="utf-8",  
                    <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with If omitted, defaults to .
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None value. If omitted, defaults to .
- **<str> encoding (optional):** The unicode encoding to use for international or special characters. For example, Microsoft applications like to use special characters for double quotes rather than the standard characters. Unicode (the default) handles these nicely. If omitted, defaults to utf-8.
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.238 TableList.save\_xml

Saves this TableList in xml format. Since the table list probably has multiple tables in it, one xml file per table is created by prepending 1, 2, 3 to the filename.

### Signature:

```
[TableList].save_xml(<str> filename,  
                    <str> line_ending="  
",  
                    <str> indent=" ",  
                    <bool> compact=False,  
                    <str> none="",  
                    <object> respect_filter=False)
```

- **<str> filename:** A file name or a file pointer to an open file. Using the file name string directly is suggested since it ensures the file is opened correctly for reading in CSV.
- **<str> line\_ending (optional):** An optional line ending to separate rows with (when compact is False) If omitted, defaults to .
- **<str> indent (optional):** The character(s) to use for indenting (when compact is False) If omitted, defaults to .
- **<bool> compact (optional):** Whether to compact the XML or make it "pretty" with whitespace If omitted, defaults to False.
- **<str> none (optional):** An optional parameter specifying what to write for cells that have the None value. If omitted, defaults to .
- **<object> respect\_filter (optional):** If omitted, defaults to False.

## 7.239 TableList.set\_changed

Sets whether the class has been changed since loading. This is not normally called by users.

**Signature:**

```
[TableList].set_changed(<object> changed)
```

- **<object> changed:**

## 7.240 `TableList.set_format`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].set_format(<object> column,  
                      <object> format=None)
```

- `<object> column`:
- `<object> format` (optional): If omitted, defaults to None.

## 7.241 `TableList.set_name`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].set_name(<object> column,  
                    <object> name)
```

- `<object> column`:
- `<object> name`:

## 7.242 `TableList.set_readonly`

Sets the read only status of this table. Tables that are read only cannot be modified. Normally, tables are initially not read only (i.e. can be modified). The only exception is tables loaded from databases, which are read only.

### Signature:

```
[TableList].set_readonly(<bool> readonly_flag=False)
```

- `<bool> readonly_flag` (optional): True or False, depending upon whether the table should be read only or not. If omitted, defaults to False.



## 7.243 `TableList.set_type`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].set_type(<object> column,  
                    <object> column_type=None,  
                    <object> format=None,  
                    <object> expression=None)
```

- **<object> column:**
- **<object> column\_type** (optional): If omitted, defaults to None.
- **<object> format** (optional): If omitted, defaults to None.
- **<object> expression** (optional): If omitted, defaults to None.

## 7.244 `TableList.structure`

This method is not supported in TableLists. Use TableArrays if this method is required.

### Signature:

```
[TableList].structure()
```

## 7.245 `TableList.view`

Opens the table list for viewing in the Picalo user interface. The resulting view allows you to page through the tables in the list. See the first example.

You can view individual tables in the list by using the [n] notation. See the second example for this notation.

### Signature:

```
[TableList].view()
```

**Example 1:**

```

1 >>> data = Table([
2 ...     ('id', int),
3 ...     ('name', unicode),
4 ... ], [
5 ...     [ 1, 'Benny' ],
6 ...     [ 2, 'Vijay' ],
7 ... ])
8 >>> tables = Grouping.stratify_by_value(data, 'id')
9 >>> tables.view()

```

**Example 2:**

```

1 >>> data = Table([
2 ...     ('id', int),
3 ...     ('name', unicode),
4 ... ], [
5 ...     [ 1, 'Benny' ],
6 ...     [ 2, 'Vijay' ],
7 ... ])
8 >>> tables = Grouping.stratify_by_value(data, 'id')
9 >>> tables[0].view()

```

## 7.246 TimeDelta

A duration in time, such as the difference between two `Datetime`s. This is often used in `Grouping.stratify_by_date` and `Grouping.summarize_by_date`. Negative numbers specify durations that go backwards in time.

**Signature:**

```

<datetime.timedelta> = TimeDelta(<int,
                                long,
                                or float> days=0,
                                <int,
                                long,
                                or float> seconds=0,
                                <int,
                                long,
                                or float> microseconds=0,
                                <int,
                                long,
                                or float> milliseconds=0,
                                <int,
                                long,

```

```

        or float> minutes=0,
        <int,
        long,
        or float> hours=0,
        <int,
        long,
        or float> weeks=0)

```

- <int, long, or float> days (optional): The number of days in this duration If omitted, defaults to 0.
- <int, long, or float> seconds (optional): The number of seconds in this duration If omitted, defaults to 0.
- <int, long, or float> microseconds (optional): The number of microseconds in this duration If omitted, defaults to 0.
- <int, long, or float> milliseconds (optional): The number of milliseconds in this duration If omitted, defaults to 0.
- <int, long, or float> minutes (optional): The number of minutes in this duration If omitted, defaults to 0.
- <int, long, or float> hours (optional): The number of hours in this duration If omitted, defaults to 0.
- <int, long, or float> weeks (optional): The number of weeks in this duration If omitted, defaults to 0.

Returns: A `datetime.timedelta` object

## 7.247 use\_progress\_indicators

Sets whether Picalo shows progress dialogs in text or GUI mode. Send `False` into this method to make Picalo quiet. Send `True` to see progress bars for operations.

### Signature:

```
use_progress_indicators(<object> show)
```

- <object> show:

## 7.248 variance

Returns the variance of the given sequence, or the default if the sequence contains zero or one items. More advanced statistical routines can be found in the `picalo.lib.stats` module.

### Signature:

```
<float> = variance(<list> sequence,  
                  <float> default=0)
```

- **<list> sequence:** A sequence of items.
- **<float> default (optional):** The default if a variance cannot be calculated. If omitted, defaults to 0.

Returns: The variance of the sequence, or the default if `len(sequence) < 2`.

# Chapter 8

## Benfords

The Benfords module performs digital analyses on data sets. In the 1930's, Benford realized that many number sets (now known to include invoice amounts and stock prices) followed a certain pattern. A 1 appeared as the first digit about 30 percent of the time. Each digit in the number has a probability associated with which number it might be.

Numbers that are created by people (who obviously don't know about Benford's Law) do not follow Benford's distribution. In recent years, Benford's Law has been used to separate values that occur naturally in business and those that are fabricated.

### 8.1 analyze

Performs a Benford's analysis on a table column. Returns a picalo table. The frequency and expected values will be 0 for items that were not analyzed (due to insufficient digits).

Important: if you ask for 2 significant digits, any input numbers that do not have two digits are ignored and not included in the results. If these numbers were not ignored, the analysis would throw errors.

#### Signature:

```
<Table> = Benfords.analyze(<Table> table,  
                           <str> col,  
                           <int> number_of_significant_digits=1)
```

- **<Table> table:** A Picalo table

- **<str> col:** The column to be analyzed.
- **<int> number\_of\_significant\_digits (optional):** The number of leading digits to use in the analysis. Higher numbers (3-5) require more data for statistical power. If omitted, defaults to 1.

Returns: A Picalo table describing the results of the analysis.

### Example 1:

```

1
2 >>> table = Table([('col000', unicode), ('col001', int), ('col002', int)], [
3 ...           ['Dan', 10, 8],
4 ...           ['Sally', 12, 12],
5 ...           ['Dan', 11, 15],
6 ...           ['Sally', 12, 14],
7 ...           ['Dan', 11, 16],
8 ...           ['Sally', 15, 15],
9 ...           ['Dan', 16, 15],
10 ...          ['Sally', 13, 14]])
11 >>> results = Benfords.analyze(table, 1, number_of_significant_digits=2)
12 >>> results.view()
13 +-----+-----+-----+-----+
14 | Number | Significant Digits | Actual Frequency | Expected Probability |
15 | Difference | | | | |
16 +-----+-----+-----+-----+
16 |      10 |      10 |          0.125 |      0.0360270497068 |
17 |      12 |      12 |          0.25 |      0.0327585353738 |
18 |      11 |      11 |          0.25 |      0.0342843373349 |
19 |      12 |      12 |          0.25 |      0.0327585353738 |
20 |      11 |      11 |          0.25 |      0.0342843373349 |
21 |      15 |      15 |          0.125 |      0.0291027478744 |
22 |      16 |      16 |          0.125 |      0.0281085963079 |
23 |      13 |      13 |          0.125 |      0.031406327064 |
24 +-----+-----+-----+-----+

```

### Example 2:

```

1 I usually add a column for the difference from Benford's expectation to the table,
2 then summarize to get an average difference per vendor, employee, etc.
3 Individual numbers will often not match Benford, but averages across several
4 numbers should match.
5
6 >>> table = Table([('col000', unicode), ('col001', int), ('col002', int)], [
7 ...           ['Dan', 10, 8],
8 ...           ['Sally', 12, 12],
9 ...           ['Dan', 11, 15],
10 ...          ['Sally', 12, 14],
11 ...          ['Dan', 11, 16],
12 ...          ['Sally', 15, 15],
13 ...          ['Dan', 16, 15],
14 ...          ['Sally', 13, 14]])
15 >>> table.append_column('ben_diff', Benfords.analyze(table, 1, 2).column(4))
16 >>> table.view()

```

```

17 |-----|
18 | col000 | col001 | col002 | ben_diff |
19 |-----|
20 | Dan    | 10     | 8       | 0.0889729502932 |
21 | Sally  | 12     | 12      | 0.217241464626  |
22 | Dan    | 11     | 15      | 0.215715662665  |
23 | Sally  | 12     | 14      | 0.217241464626  |
24 | Dan    | 11     | 16      | 0.215715662665  |
25 | Sally  | 15     | 15      | 0.0958972521256 |
26 | Dan    | 16     | 15      | 0.0968914036921 |
27 | Sally  | 13     | 14      | 0.093593672936  |
28 |-----|
29 >>> results = Grouping.summarize_by_value(table, 'col000',
30 ...      ben_avg="sum(group['ben_diff']) / len(group)")
31 >>> results.view()
32 |-----|
33 | StartKey | EndKey | ben_avg |
34 |-----|
35 | Dan      | Dan    | 0.154323919829 |
36 | Sally    | Sally  | 0.155993463579 |
37 |-----|

```

## 8.2 calc\_benford

Helper function that codes benford's actual formula The generalized formula was found at <http://www.mathpages.com/home/kmath302/kmath302.htm> This method calculates the probability at a given digit is in a given position.

### Signature:

```

<float> = Benfords.calc_benford(<int> position,
                                <int> digit,
                                <int> base=10)

```

- **<int> position:** The position in the number (0=first digit, 1=second digit, ...)
- **<int> digit:** The actual number (0,1,2,3,4,5,6,7,8,9) this digit is
- **<int> base (optional):** The number base. Optional (default is 10) If omitted, defaults to 10.

Returns: The Benford probability (the percentage of the time this position will have this digit).

### 8.3 `get_expected`

Calculates Benford's expected probability for a given number to a certain number of digits. For example, given the number 1234, calculate the combined probability that a 1 appears as the first digit, a 2 appears as the second digit, and so forth, to the number of desired significant digits.

**Signature:**

```
<float> = Benfords.get_expected(<float> number,  
                                <int> number_of_significant_digits=1)
```

- **<float> number:** The number (1234 in the example) to calculate the probability for
- **<int> number\_of\_significant\_digits (optional):** The number of positions to use for the probability. In the example, a value of 2 means to calculate the probability for the number 12. If omitted, defaults to 1.

Returns: The expected frequency of this number according to Benford's Law.



# Chapter 9

## Crosstable

The Crosstable module creates crosstables of data. It is similar to Excel's powerful PivotTable function. Crosstabbing takes data that is in database format and converts it to spreadsheet format, usually with summarization functions like sum, average, etc.

The primary function of this module is `pivot()`. The other functions are more advanced functions that show different levels of detail from the crosstabbing process.

Pivoting extremely large tables can take some time, depending upon your processor speed and amount of memory. Assuming you are running Picalo in GUI mode, the functions will show a progress bar.

### 9.1 pivot

Crosstables a Picalo table into a new table. This routine is similar to Excel's PivotTable feature, but it is quite a bit more powerful (although not as easy to use).

The pivot function flattens the results into a two-dimensional table, with a column for each expression of each data field. Of all the pivot functions in the Crosstable module, this is the most like Excel's pivot table feature.

There are no arbitrary limits on table size in Picalo, so you can pivot tables with many values (resulting in a significant number of rows and columns). The only limit on the number of resulting rows and columns is your memory.

The first example shows a simple pivot on a single `col_field`, `row_field`, and single expression. The second example shows a more complex pivot

with multiple fields and expressions. Note that the second example has some errors because the average function is being run on cells that have no values (divide by zero).

### Signature:

```
<Table> = Crosstable.pivot(<Table or TableArray> table,
                           <str/list> col_fields,
                           <str/list> row_fields,
                           <str/list> expressions,
                           <str> delimiter="_")
```

- **<Table or TableArray> table:** The Picalo table that will be crosstabled
- **<str/list> col\_fields:** A single field name or a list containing the field names of the fields used to group columns.
- **<str/list> row\_fields:** A single field name or a list containing the field names of the fields used to group rows
- **<str/list> expressions:** A single expression or a list containing expressions used to do calculations on groupings; when the expression is evaluated, the keyword 'group' denotes the matching records for each cell
- **<str> delimiter (optional):** The character(s) to use to combine field values and expressions (for row and column headers) If omitted, defaults to \_.

Returns: A new table containing the crosstabled results. The last column and last row of the table contain the row and column totals

### Example 1:

```
1 >>> # create a test table
2 >>> data = Table([
3 ...   ('Region', unicode),
4 ...   ('Product', unicode),
5 ...   ('Salesperson', unicode),
6 ...   ('Customer', unicode),
7 ...   ('Amount', int),
8 ... ], [
9 ...   [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ...  [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
```

```

11 ... [ 'Rural', 'Mice',      'Mollie', 'Brian', 900 ],
12 ... [ 'City',  'Computers', 'Danny', 'Faiz', 300 ],
13 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ... [ 'City',  'Mice',      'Danny', 'Brian', 100 ],
16 ... [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ... [ 'City',  'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot(data, 'Salesperson', 'Product', 'sum(group["Amount"])')
21 >>> ret.view()

```

	Pivot	Danny	Mollie	Totals
Computers		700	500	1200
Mice		100	900	1000
Monitors		700	1200	1900
Totals		1500	2600	4100

**Example 2:**

```

1 >>> # create a test table
2 >>> data = Table([
3 ... ('Region', unicode),
4 ... ('Product', unicode),
5 ... ('Salesperson', unicode),
6 ... ('Customer', unicode),
7 ... ('Amount', int),
8 ... ], [
9 ... [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ... [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
11 ... [ 'Rural', 'Mice', 'Mollie', 'Brian', 900 ],
12 ... [ 'City', 'Computers', 'Danny', 'Faiz', 300 ],
13 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ... [ 'City', 'Mice', 'Danny', 'Brian', 100 ],
16 ... [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ... [ 'City', 'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot(data, ['Salesperson', 'Customer'], ['Product', 'Region'], ['sum(group["Amount"])'])
21 >>> ret.view()

```

	Pivot	Danny_Brian_expr1		Danny_Brian_expr2		Danny_Faiz_expr1		Danny_Faiz_expr2		Totals_expr2
Computers_City		0	0	0	0	300	0	0	0	300
Computers_Rural		0	0	0	0	0	0	0	0	0
Mice_City		0	0	0	0	0	0	0	0	0
Mice_Rural		0	0	0	0	0	0	0	0	0
Monitors_City		0	0	0	0	0	0	0	0	0
Monitors_Rural		0	0	0	0	0	0	0	0	0
Totals		0	0	0	0	0	0	0	0	0

	0			0			0			0		
	0		100			100						
28		Mice_Rural				0			0			0
	0			0			0			900		
	900			0			0			0		
	0		900			900						
29		Monitors_City				0			0			0
	0			700			700			0		
	0			0			0			0		
	0		700			700						
30		Monitors_Rural				0			0			0
	0			0			0			0		
	0			200			200			1000		
	500.0			1200			400.0					
31		Totals				100			100			300
	300			1100			550.0			900		
	900			700			350.0			1000		
	500.0			4100			455.555555556					
32	+		+		+			+			+	

## 9.2 pivot\_map

Matches all unique combinations of values in `col_fields` and `row_fields` with expressions run on the records that containing those unique values.

This function is very similar to `Grouping.summarize_by_value`, only it is formatted in a way to make crosstabling possible.

This is an advanced function that is used internally during the pivot technique. It is provided for advanced users who want to access the detail records during the crosstabling process. It is one step in the process beyond `pivot_map_detail`.

Most users should use `Crosstable.pivot` instead as it is more like Excel's pivot function.

### Signature:

```
<dict> = Crosstable.pivot_map(<Table> table,
                              <str/list> col_fields,
                              <str/list> row_fields,
                              <str/list> expressions)
```

- **<Table> table:** The Picalo table that will be crosstabled
- **<str/list> col\_fields:** A single field name or a list containing the field names of the fields used to group columns.

- **<str/list> row\_fields:** A single field name or a list containing the field names of the fields used to group rows
- **<str/list> expressions:** A single expression or a list containing expressions used to do calculations on groupings; when the expression is evaluated, the keyword 'group' denotes the matching records for each cell

Returns: A dictionary of each unique key (made up of row and column combinations) mapped to their matching records.

### Example:

```

1 >>> # create a test table
2 >>> data = Table([
3 ...   ('Region', unicode),
4 ...   ('Product', unicode),
5 ...   ('Salesperson', unicode),
6 ...   ('Customer', unicode),
7 ...   ('Amount', int),
8 ... ], [
9 ...   [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ...   [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
11 ...   [ 'Rural', 'Mice', 'Mollie', 'Brian', 900 ],
12 ...   [ 'City', 'Computers', 'Danny', 'Faiz', 300 ],
13 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ...   [ 'City', 'Mice', 'Danny', 'Brian', 100 ],
16 ...   [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ...   [ 'City', 'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot_map(data, 'Salesperson', 'Product', 'sum(group["Amount"])')
21 >>> ret
22 {
23   ('Monitors', 'Mollie'): (1200,),
24   ('Computers', 'Mollie'): (500,),
25   ('Mice', 'Danny'): (100,),
26   ('Computers', 'Danny'): (700,),
27   ('Mice', 'Mollie'): (900,),
28   ('Monitors', 'Danny'): (700,),
29 }
```

## 9.3 pivot\_map\_detail

Matches all unique combinations of values in col\_fields and row\_fields with Tables that contain only the records with those values. This function is a "mega-select" function that is the basis of the crosstabbing technique.

Use this function if you just want to separate a Table in a number of subtables – one for each unique combination of `col_fields` and `row_fields`.

This function is very similar to `Grouping.stratify_by_value`, only it is formatted in a way to make crosstabling possible.

This is an advanced function that is used internally during the pivot technique. It is provided for advanced users who want to access the detail records during the crosstabling process. It is the first step performed in a crosstable.

Most users should use `Crosstable.pivot` instead as it is more like Excel's pivot function.

### Signature:

```
<dict> = Crosstable.pivot_map_detail(<Table> table,
                                     <str/list> col_fields,
                                     <str/list> row_fields)
```

- **<Table> table:** The Picalo table that will be crosstabled
- **<str/list> col\_fields:** A single field name or a list containing the field names of the fields used to group columns.
- **<str/list> row\_fields:** A single field name or a list containing the field names of the fields used to group rows

Returns: A dictionary of each unique key (made up of row and column combinations) mapped to Tables containing their matching records.

### Example:

```
1 >>> # create a test table
2 >>> data = Table([
3 ...   ('Region', unicode),
4 ...   ('Product', unicode),
5 ...   ('Salesperson', unicode),
6 ...   ('Customer', unicode),
7 ...   ('Amount', int),
8 ... ], [
9 ...   [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ...   [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
11 ...   [ 'Rural', 'Mice', 'Mollie', 'Brian', 900 ],
12 ...   [ 'City', 'Computers', 'Danny', 'Faiz', 300 ],
13 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ...   [ 'City', 'Mice', 'Danny', 'Brian', 100 ]],
```

```

16 ... [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ... [ 'City', 'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot_map_detail(data, 'Salesperson', 'Product')
21 >>> ret
22 {
23   ('Mollie', 'Computers'): <Table: 1 rows x 5 cols>,
24   ('Danny', 'Computers'): <Table: 2 rows x 5 cols>,
25   ('Mollie', 'Monitors'): <Table: 3 rows x 5 cols>,
26   ('Danny', 'Monitors'): <Table: 1 rows x 5 cols>,
27   ('Danny', 'Mice'): <Table: 1 rows x 5 cols>,
28   ('Mollie', 'Mice'): <Table: 1 rows x 5 cols>
29 }

```

## 9.4 pivot\_table

Crosstables a Picalo table into a new table. This routine is similar to Excel's PivotTable feature, but it is quite a bit more powerful (although not as easy to use).

This version creates does not flatten results to a two-dimensional table like the pivot function. It is a more advanced way of pivoting than the pivot function because it creates a list of expressions results for each cell. In other words, the results table is not normalized, but contains lists within each cell that contain the results. When a single col\_field, row\_field, and expression is given, this function produces the exact same results as pivot().

This way of pivoting is useful if you want a single column for each col field value and a single row for each row field value. Even if you provide multiple expressions and/or multiple data fields, you'll still only get one col/row match for a given value set. The multiple expressions on the multiple data fields will be contained in a list in the cell for each row/col match.

There are no arbitrary limits on table size in Picalo, so you can pivot tables with many values (resulting in a significant number of rows and columns). The only limit on the number of resulting rows and columns is your memory.

The first example shows a simple pivot on a single col\_field, row\_field, and single expression. The second example shows a more complex pivot with multiple fields and expressions. Note that the second example has some errors because the average function is being run on cells that have no values (divide by zero).

**Signature:**

```
<Table> = Crosstable.pivot_table(<Table or TableArray> table,
                                <str/list> col_fields,
                                <str/list> row_fields,
                                <str/list> expressions)
```

- **<Table or TableArray> table:** The Picalo table that will be crosstabled
- **<str/list> col\_fields:** A single field name or a list containing the field names of the fields used to group columns.
- **<str/list> row\_fields:** A single field name or a list containing the field names of the fields used to group rows
- **<str/list> expressions:** A single expression or a list containing expressions used to do calculations on groupings; when the expression is evaluated, the keyword 'group' denotes the matching records for each cell

Returns: A new table containing the crosstabled results. The last column and last row of the table contain the row and column totals

### Example 1:

```
1 >>> # create a test table
2 >>> data = Table([
3 ...   ('Region', unicode),
4 ...   ('Product', unicode),
5 ...   ('Salesperson', unicode),
6 ...   ('Customer', unicode),
7 ...   ('Amount', int),
8 ... ], [
9 ...   [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ...   [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
11 ...   [ 'Rural', 'Mice', 'Mollie', 'Brian', 900 ],
12 ...   [ 'City', 'Computers', 'Danny', 'Faiz', 300 ],
13 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ...   [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ...   [ 'City', 'Mice', 'Danny', 'Brian', 100 ],
16 ...   [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ...   [ 'City', 'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot_table(data, 'Salesperson', 'Product', 'sum(group["Amount"])')
21 >>> ret.view()
```

	Pivot	Danny	Mollie	Totals
Computers		700	500	1200
Mice		100	900	1000



27		Monitors		700		1200		1900	
28		Totals		1500		2600		4100	
29									

**Example 2:**

```

1 >>> # create a test table
2 >>> data = Table([
3 ... ('Region', unicode),
4 ... ('Product', unicode),
5 ... ('Salesperson', unicode),
6 ... ('Customer', unicode),
7 ... ('Amount', int),
8 ... ], [
9 ... [ 'Rural', 'Computers', 'Mollie', 'Faiz', 500 ],
10 ... [ 'City', 'Monitors', 'Danny', 'Sheng', 700 ],
11 ... [ 'Rural', 'Mice', 'Mollie', 'Brian', 900 ],
12 ... [ 'City', 'Computers', 'Danny', 'Faiz', 300 ],
13 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
14 ... [ 'Rural', 'Monitors', 'Mollie', 'Sheng', 500 ],
15 ... [ 'City', 'Mice', 'Danny', 'Brian', 100 ],
16 ... [ 'Rural', 'Monitors', 'Mollie', 'Faiz', 200 ],
17 ... [ 'City', 'Computers', 'Danny', 'Sheng', 400 ],
18 ... ])
19 >>> # perform the pivot
20 >>> ret = Crosstable.pivot_table(data, ['Salesperson', 'Customer'], ['Product', 'Region'], ['sum(gro
21 >>> ret.view()

```

22									
23									
		Pivot		( 'Danny', 'Brian' )		( 'Danny', 'Faiz' )		( 'Danny', 'Sheng' )	
		Totals							
24									
25		( 'Computers', 'City' )		(0, 0)		(300, 300)		(400, 400)	
		(0, 0)		(0, 0)		(0, 0)		(700, 350.0)	
26		( 'Computers', 'Rural' )		(0, 0)		(0, 0)			
		(0, 0)		(0, 0)		(500, 500)		(0, 0)	
		(500, 500)							
27		( 'Mice', 'City' )		(100, 100)		(0, 0)			
		(0, 0)		(0, 0)		(0, 0)		(0, 0)	
		(100, 100)							
28		( 'Mice', 'Rural' )		(0, 0)		(0, 0)			
		(0, 0)		(900, 900)		(0, 0)		(0, 0)	
		(900, 900)							
29		( 'Monitors', 'City' )		(0, 0)		(0, 0)		(700, 700)	
		(0, 0)		(0, 0)		(0, 0)		(700, 700)	
30		( 'Monitors', 'Rural' )		(0, 0)		(0, 0)			
		(0, 0)		(0, 0)		(200, 200)		(1000, 500.0)	
		(1200, 400.0)							
31		Totals		(100, 100)		(300, 300)		(1100, 550.0)	
		(900, 900)		(700, 350.0)		(1000, 500.0)		(4100, 455.55555555555554)	
32									

# Chapter 10

## Database

This module provides easy access to DB-API 2.0 tables. It is a decorator for connections and cursors. Cursor results can be accessed efficiently via field name or index.

Right now ODBC data sources are the primary mechanism to access databases. Set up ODBC in your operating system and call the `Odbc-Connection` method to create a connection. In addition, direct connections to MySQL and PostgreSQL are supported (without the need for an ODBC DSN setup).

The primary use of this module is `table()`, which returns a standard Picalo table. Other methods are provided for advanced users who want more control over the database connection.

Most users only need to learn how to 1) create Connection objects, and 2) run `conn.table()` to query database tables.

The module also provides classes to ease the creation of insert, update, and select queries.

### 10.1 MySQLConnection

Opens a database connection to a MySQL database using the MySQLdb driver.

#### Signature:

```
Database.MySQLConnection(<str> database,  
                          <str> username="None",
```

```
<str> password="None",  
<str> host="None",  
<int> port=None)
```

- **<str> database:** The database name to connect to.
- **<str> username (optional):** Your username for the connection. If omitted, defaults to None.
- **<str> password (optional):** Your password for the connection. If omitted, defaults to None.
- **<str> host (optional):** The server hostname or IP address. If omitted, defaults to None.
- **<int> port (optional):** The server port to connect on. If omitted, defaults to None.

## 10.2 OdbcConnection

Opens a database connection to an ODBC database using the PyODBC driver.

### Signature:

```
Database.OdbcConnection(<str> dsn_name,  
                        <str> username="None",  
                        <str> password="None")
```

- **<str> dsn\_name:** The DSN string to your database (as defined in the control panel)
- **<str> username (optional):** Your username for the connection. If omitted, defaults to None.
- **<str> password (optional):** Your password for the connection. If omitted, defaults to None.

## 10.3 OracleConnection

Opens a database connection to an Oracle database using the cx\_Oracle driver.

**Signature:**

```
Database.OracleConnection(<str> dsn,  
                           <str> username="None",  
                           <str> password="None")
```

- **<str> dsn:** The DSN string to your database
- **<str> username (optional):** Your username for the connection. If omitted, defaults to None.
- **<str> password (optional):** Your password for the connection. If omitted, defaults to None.

## 10.4 PostgreSQLConnection

Opens a database connection to a PostgreSQL database using the psycopg2 driver.

**Signature:**

```
Database.PostgreSQLConnection(<str> database,  
                              <str> username="None",  
                              <str> password="None",  
                              <str> host="None",  
                              <int> port=None)
```

- **<str> database:** The database name to connect to.
- **<str> username (optional):** Your username for the connection. If omitted, defaults to None.
- **<str> password (optional):** Your password for the connection. If omitted, defaults to None.

- `<str> host` (optional): The server hostname or IP address. If omitted, defaults to `None`.
- `<int> port` (optional): The server port to connect on. If omitted, defaults to `None`.

## 10.5 PyGreSQLConnection

Opens a database connection to a PostgreSQL database using the PyGreSQL driver.

### Signature:

```
Database.PyGreSQLConnection(<str> database,  
                             <str> username="None",  
                             <str> password="None",  
                             <str> host="None",  
                             <int> port=None)
```

- **`<str> database`**: The database name to connect to.
- `<str> username` (optional): Your username for the connection. If omitted, defaults to `None`.
- `<str> password` (optional): Your password for the connection. If omitted, defaults to `None`.
- `<str> host` (optional): The server hostname or IP address. If omitted, defaults to `None`.
- `<int> port` (optional): The server port to connect on. If omitted, defaults to `None`.

## 10.6 SqliteConnection

Opens a database connection to an SQLite database using the built-in `sqlite3` driver. SQLite is a lightweight, disk-based database that doesn't require a separate server. It is an excellent choice for small applications where you want to use SQL but don't need a "real" database like MySQL, PostgreSQL,

or Oracle. Since it comes with Picalo, it's ready to go immediately – nothing is required except a Picalo install.

**Signature:**

`Database.SqliteConnection(<str> dirname)`

- **<str> dirname:** The directory to store database files in, or `":memory:"` to keep everything in memory.

# Chapter 11

## Financial

Encodes financial ratios used for analysis of financial statements. While most of these ratios are easy to code, this module provides a home for their standard coding.

These functions are useful in expressions used throughout Picalo.

### 11.1 `asset_turnover`

Returns the asset turnover ratio

**Signature:**

```
Financial.asset_turnover(<object> sales,  
                        <object> beg_total_assets,  
                        <object> end_total_assets)
```

- `<object> sales:`
- `<object> beg_total_assets:`
- `<object> end_total_assets:`

### 11.2 `current_ratio`

Returns the current ratio

**Signature:**

```
Financial.current_ratio(<object> current_assets,  
                        <object> current_liabilities)
```

- <object> current\_assets:
- <object> current\_liabilities:

## 11.3 debt\_to\_equity

Returns the debt to equity ratio

**Signature:**

```
Financial.debt_to_equity(<object> total_liabilities,  
                        <object> total_stockholders_equity)
```

- <object> total\_liabilities:
- <object> total\_stockholders\_equity:

## 11.4 earnings\_per\_share

Returns the earnings per share

**Signature:**

```
Financial.earnings_per_share(<object> net_income,  
                             <object> number_of_shares)
```

- <object> net\_income:
- <object> number\_of\_shares:

## 11.5 inventory\_turnover

Returns the inventory turnover ratio

**Signature:**



```
Financial.inventory_turnover(<object> cost_of_goods_sold,  
                             <object> beg_inventory,  
                             <object> end_inventory)
```

- <object> cost\_of\_goods\_sold:
- <object> beg\_inventory:
- <object> end\_inventory:

## 11.6 net\_working\_capital

Returns the net working capital ratio

**Signature:**

```
Financial.net_working_capital(<object> current_assets,  
                              <object> current_liabilities,  
                              <object> total_assets)
```

- <object> current\_assets:
- <object> current\_liabilities:
- <object> total\_assets:

## 11.7 price\_earnings

Returns the PE (price earnings) ratio

**Signature:**

```
Financial.price_earnings(<object> share_market_price,  
                         <object> net_income,  
                         <object> number_of_shares)
```

- <object> share\_market\_price:
- <object> net\_income:
- <object> number\_of\_shares:

## 11.8 profit\_margin

Returns the profit margin

**Signature:**

```
Financial.profit_margin(<object> net_income,  
                       <object> sales)
```

- <object> net\_income:
- <object> sales:

## 11.9 quick\_ratio

Returns the quick ratio

**Signature:**

```
Financial.quick_ratio(<object> current_assets,  
                     <object> inventory,  
                     <object> current_liabilities)
```

- <object> current\_assets:
- <object> inventory:
- <object> current\_liabilities:

## 11.10 return\_on\_assets

Returns the return on assets

**Signature:**

```
Financial.return_on_assets(<object> net_income,  
                          <object> beg_total_assets,  
                          <object> end_total_assets)
```

- <object> net\_income:
- <object> beg\_total\_assets:
- <object> end\_total\_assets:

## 11.11 `return_on_common_equity`

Returns the return on common equity

**Signature:**

```
Financial.return_on_common_equity(<object> net_income,  
                                  <object> beg_common_stockholders_equity,  
                                  <object> end_common_stockholders_equity)
```

- `<object> net_income:`
- `<object> beg_common_stockholders_equity:`
- `<object> end_common_stockholders_equity:`

## 11.12 `return_on_equity`

Returns the return on equity

**Signature:**

```
Financial.return_on_equity(<object> net_income,  
                           <object> beg_stockholders_equity,  
                           <object> end_stockholders_equity)
```

- `<object> net_income:`
- `<object> beg_stockholders_equity:`
- `<object> end_stockholders_equity:`

# Chapter 12

## Grouping

The Grouping module contains functions that stratify and summarize records in different ways. Grouping is a basic fraud detection method that helps generate norms and compare records against those norms. Further, it splits data sets into individual groups that should be analyzed separately.

The Grouping module only stratifies a table into many tables. It doesn't do any summarization of values. All detail data are still in the tables.

The summary routines not only stratify data on key, but also summarize the detail tables using sum, mean, and other statistical routines. You get only one table returned from Summarize. Stratifying gives the intermediate result – the full detail in many tables.

For those familiar with SQL, the summarize routines are similar to the GROUP BY keyword. Groups are collapsed into summary records and a single table is returned. The stratifying functions are similar to running one query to retrieve all unique key values, followed by a query to retrieve the records that match each key value. The result is many tables (one per unique key value).

### 12.1 stratify

Stratifies a Picalo table into a specified number of sub-tables. The table should be sorted appropriately before this function is called. This function does not modify the underlying table.

The start and end record indices are recorded in table.startvalue and table.endvalue. See the example for more information.

**Signature:**

```
<TableArray> = Grouping.stratify(<Table> table,
                                <int> number_of_groups)
```

- **<Table> table:** The table to be stratified
- **<int> number\_of\_groups:** The number of sub-tables to create

Returns: A list of new tables

**Example:**

```
1 >>> table = Table([('col000', int), ('col001', int)], [[1,1],[2,2],[3,3]])
2 >>> groups = Grouping.stratify(table, 2)
3 >>> print groups[0].startvalue, groups[0].endvalue
4 0, 1
5 >>> groups[0].view()
6 +-----+
7 | col000 | col001 |
8 +-----+
9 |      1 |      1 |
10 |      2 |      2 |
11 +-----+
12 >>> print groups[1].startvalue, groups[1].endvalue
13 3, 3
14 >>> groups[1].view()
15 +-----+
16 | col000 | col001 |
17 +-----+
18 |      3 |      3 |
19 +-----+
```

## 12.2 stratify\_by\_date

Stratifies the table rows into specific groups by date ranges. This function is useful to split a table into ranges such as two-week periods (useful for analyzing timecard and invoice databases).

Aging can also be done by sorting the table in reverse (newest to oldest dates) and using a negative duration.

The start and end values of each group are recorded in the table object since they may be different than the actual start and end column values. See the example for more information.

See the `base.Calendar` module for more information about time durations.

**Signature:**

```
<TableArray> = Grouping.stratify_by_date(<Table> table,
                                         <str> col,
                                         <int> duration)
```

- **<Table> table:** The table to be stratified
- **<str> col:** The column to stratify by.
- **<int> duration:** The number of days or seconds to put into each group.

Returns: A list of new tables, each containing rows from table that were stratified together

### Example:

```
1 >>> table = Table([('col000', DateTime)], [
2 ...   [DateTime(2000,1,1)],
3 ...   [DateTime(2000,1,13)],
4 ...   [DateTime(2000,1,14)],
5 ...   [DateTime(2000,1,15)]
6 ... ])
7 >>> groups = Grouping.stratify_by_date(table, 0, DateDelta(7))
8 >>> print groups[0].startvalue, groups[0].endvalue
9 2000-01-01 00:00:00, 2000-01-08 00:00:00
10 >>> groups[0].view()
11 +-----+
12 |          col000          |
13 +-----+
14 | 2000-01-01 00:00:00.00 |
15 +-----+
16 >>> print groups[1].startvalue, groups[1].endvalue
17 2000-01-08 00:00:00, 2000-01-15 00:00:00
18 >>> groups[1].view()
19 +-----+
20 |          col000          |
21 +-----+
22 | 2000-01-13 00:00:00.00 |
23 | 2000-01-14 00:00:00.00 |
24 +-----+
25 >>> print groups[2].startvalue, groups[2].endvalue
26 2000-01-15 00:00:00, 2000-01-22 00:00:00
27 >>> groups[2].view()
28 +-----+
29 |          col000          |
30 +-----+
31 | 2000-01-15 00:00:00.00 |
32 +-----+
```

## 12.3 stratify\_by\_expression

Stratifies a table based upon the return value from an expression. For each record in the table, the expression is evaluated with the following variables:

1. `startrecord` => the starting record of the current group.
2. `record` => the current record being evaluated. If the expression evaluates to `True`, `rec` is placed in a new group and becomes `startrec`. If the expression evaluates to `False`, `rec` is placed in the current group.

The start and end record indices are recorded in `table.startvalue` and `table.endvalue`. See the example for more information.

```
Example (starts a new group on each odd value in column 1): >>> table1
= Table(['col000', int]), [[1],[2],[3],[4]]) >>> groups = Grouping.stratify_by_expression(table1,
"record[0] % 2.0 == 1.0") >>> print groups[0].startvalue, groups[0].endvalue
0, 1 >>> groups[0].view() +-----+ -- col000 -- +-----+ -- 1 -- -- 2
-- +-----+ >>> print groups[1].startvalue, groups[1].endvalue 3, 4 >>>
groups[1].view() +-----+ -- col000 -- +-----+ -- 3 -- -- 4 -- +-----+
```

### Signature:

```
<TableArray> = Grouping.stratify_by_expression(<Table> table,
                                              <str> expression)
```

- **<Table> table:** The table to be stratified
- **<str> expression:** An expression that evaluates the current record and returns whether a new group should be started

Returns: A list of new tables, each containing rows from table that were grouped together

## 12.4 stratify\_by\_step

Stratifies a table based upon the value of `col`. Each time the value of `col` jumps `> step`, a new group is started.

The table should be sorted correctly \*before\* this method is called. This method simply runs through the table records sequentially.

The start and end values of each group are recorded in the table object since they may be different than the actual start and end column values. See the example for more information.

Records are stratified where  $\text{startvalue} \geq \text{record}[\text{col}] < \text{endvalue}$ .

### Signature:

```
<TableArray> = Grouping.stratify_by_step(<Table> table,
                                         <str> col,
                                         <int> step)
```

- **<Table> table:** The table to be stratified
- **<str> col:** The column to use to step by
- **<int> step:** The step value that starts a new group

Returns: A list of new tables, each containing rows from table that were stratified together

### Example:

```
1 >>> table1 = Table([( 'col000', int), ( 'col001', int)], [[1,1], [2,2], [5.9,3], [6,1], [8,2], [16,1],
2 >>> groups = Grouping.stratify_by_step(table1, 0, 5)
3 >>> print groups[0].startvalue, groups[0].endvalue
4 1, 6
5 >>> groups[0].view()
6 +-----+
7 | col000 | col001 |
8 +-----+
9 |      1 |      1 |
10 |      2 |      2 |
11 |     5.9 |      3 |
12 +-----+
13 >>> print groups[1].startvalue, groups[1].endvalue
14 6, 11
15 >>> groups[1].view()
16 +-----+
17 | col000 | col001 |
18 +-----+
19 |      6 |      1 |
20 |      8 |      2 |
21 +-----+
22 >>> print groups[2].startvalue, groups[2].endvalue
23 11, 16
24 >>> groups[2].view()
25 +-----+
26 | col000 | col001 |
27 +-----+
28 +-----+
29 >>> print groups[3].startvalue, groups[3].endvalue
30 16, 21
31 >>> groups[3].view()
32 +-----+
```



33		col000		col001	
34	+		+		+
35		16		1	
36	+		+		+

## 12.5 stratify\_by\_value

Stratifies a Picalo table by composite key (combination of values in the cols columns). A new table is created for each unique composite key, resulting in a list of tables. This function does not modify the underlying table.

Each key is recorded as the start and end values of each group. See the example for more information.

### Signature:

```
<TableArray> = Grouping.stratify_by_value(<Table> table)
```

- **<Table> table:** The table to be stratified

Returns: A list of new tables, each table containing rows with the same key values.

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], [[ 'Dan', 10], [ 'Sally', 10], [ 'Dan', 11]])
2 >>> groups = Grouping.stratify_by_value(table, 'col000')
3 >>> groups[0].view()
4 +-----+
5 | col000 | col001 |
6 +-----+
7 |  Dan   |    10   |
8 |  Dan   |    11   |
9 +-----+
10 >>> groups[1].view()
11 +-----+
12 | col000 | col001 |
13 +-----+
14 |  Sally |    10   |
15 +-----+
```

## 12.6 summarize

Summarizes a sequence of groups by evaluating a series of expressions on each group. The result is a single table with one row representing each group in groups. This is analogous to the SQL GROUP BY command.

Each item in expressions should be an expression that summarizes a group. It must evaluate to a single value summarizing the entire table. Most expressions will probably only summarize a single column as is done in the example. Use the `Table['colname']` method to get the desired column.

### Signature:

```
<Table> = Grouping.summarize(<TableArray> groups)
```

- **<TableArray> groups:** A TableArray of groups, probably created by one of the `stratify_by_...` routines.

Returns: A single table containing the summaries of the groups.

### Example:

```
1 >>> from picalo.lib import stats
2 >>> table = Table([('col000', int), ('col001', int), ('col002', int)], [[1,1,1],[1,1,2],[2,1,2],[2,1,2]]
3 >>> groups = Grouping.stratify_by_value(table, 0, 1)
4 >>> summary = Grouping.summarize(groups, sum="sum(group['col002'])", avg="stats.mean(group['col002'])")
5 >>> summary.view()
```

	StartValue	EndValue	sum	avg
9	(1, 1)	(1, 1)	3	1.5
10	(2, 1)	(2, 1)	5	2.5

## 12.7 summarize\_by\_date

Summarizes the table rows into specific groups by date ranges. The function then summarizes the list of groups by running a series of functions on each group. The result is single table with one row representing each group. This is analogous to the SQL GROUP BY command. This function does not modify the underlying table.

Aging can also be done by sorting the table in reverse (newest to oldest dates) and using a negative `num_days_in_groups`.

Each item in expressions should be an expression that summarizes a group. It must evaluate to a single value summarizing the entire table. Most expressions will probably only summarize a single column as is done in the example. Use the `Table['colname']` method to get the desired column.

### Signature:

```
<Table> = Grouping.summarize_by_date(<Table> table,
                                     <str> col,
                                     <int> num_days_in_groups)
```

- **<Table> table:** The table to be summarized
- **<str> col:** The column to stratify by.
- **<int> num\_days\_in\_groups:** The number of days or seconds to put into each group.

Returns: A single table containing the summaries of the groups.

### Example:

```
1 >>> table = Table([( 'col000' , DateTime), ( 'col001' , int), ( 'col002' , int)], [
2 ...   [DateTime(2000,1,1)],
3 ...   [DateTime(2000,1,13)],
4 ...   [DateTime(2000,1,14)],
5 ...   [DateTime(2000,1,15)]
6 ... ])
7 >>> summary = Grouping.summarize_by_date(table, 0, TimeDelta(7), first="group[0][0]", last="group[0][1]")
8 >>> summary.view()
```

	StartValue	EndValue	last	first
12	2000-01-01 00:00:00	2000-01-08 00:00:00	2000-01-01 00:00:00	2000-01-01 00:00:00
13	2000-01-08 00:00:00	2000-01-15 00:00:00	2000-01-14 00:00:00	2000-01-13 00:00:00
14	2000-01-15 00:00:00	2000-01-22 00:00:00	2000-01-15 00:00:00	2000-01-15 00:00:00

## 12.8 summarize\_by\_expression

Summarizes a table based upon the return value from expressions. The function then summarizes the list of groups by running a series of functions on each group. The result is single table with one row representing each group. This is analogous to the SQL GROUP BY command. This function does not modify the underlying table.

Each item in expressions should be an expression that summarizes a group. It must evaluate to a single value summarizing the entire table. Most expressions will probably only summarize a single column as is done in the example. Use the Table['colname'] method to get the desired column.

### Signature:

```
<Table> = Grouping.summarize_by_expression(<Table> table,  
                                           <str> stratifying_expression)
```

- **<Table> table:** The table to be summarized
- **<str> stratifying\_expression:** An expression that evaluates the current record and returns whether a new group should be started

Returns: A single table containing the summaries of the groups.

## 12.9 summarize\_by\_step

Summarizes a table based upon the value of col. Each time the value of col jumps > step, a new group is started. The function then summarizes the list of groups by running a series of functions on each group. The result is single table with one row representing each group. This is analogous to the SQL GROUP BY command. This function does not modify the underlying table.

Each item in expressions should be an expression that summarizes a group. It must evaluate to a single value summarizing the entire table. Most expressions will probably only summarize a single column as is done in the example. Use the Table[colname] method to get the desired column.

**Signature:**

```
<Table> = Grouping.summarize_by_step(<Table> table,  
                                   <str> col,  
                                   <float> step)
```

- **<Table> table:** The table to be summarized
- **<str> col:** The column to stratify by.
- **<float> step:** The step value that starts a new group

Returns: A single table containing the summaries of the groups.

**Example:**

```

1 >>> from picalo.lib import stats
2 >>> table1 = Table([('col000', int), ('col001', int)], [[1,1], [2,2], [5.9,3], [6,1], [8,2], [16,1]]
3 >>> summary = Grouping.summarize_by_step(table1, 0, 5, count="len(group)")
4 >>> summary.view()
5 +-----+-----+-----+
6 | StartValue | EndValue | count |
7 +-----+-----+-----+
8 |          1 |         6 |      3 |
9 |          6 |        11 |      2 |
10 |         11 |        16 |      0 |
11 |         16 |        21 |      1 |
12 +-----+-----+-----+

```

## 12.10 summarize\_by\_value

Stratifies a Picalo table by composite key (combination of values in the `col_list` columns). A new table is created for each unique composite key, resulting in a list of tables. The function then summarizes the list of groups by running a series of expressions on each group. The result is single table with one row representing each group. This is analogous to the SQL GROUP BY command. This function does not modify the underlying table.

Each item in expressions should be an expression that summarizes a group. It must evaluate to a single value summarizing the entire table. Most expressions will probably only summarize a single column as is done in the example. Use the `Table['colname']` method to get the desired column.

### Signature:

```
<Table> = Grouping.summarize_by_value(<Table> table)
```

- **<Table> table:** The table to be summarized

Returns: A single table containing the summaries of the groups.

### Example:

```

1 >>> from picalo import *
2 >>> from picalo.lib import stats
3 >>> table = Table([('col000', unicode), ('col001', int), ('col002', int)], [
4 ...     ['Dan', 10, 8],
5 ...     ['Sally', 12, 12],
6 ...     ['Dan', 11, 15],
7 ...     ['Sally', 12, 14],
8 ...     ['Dan', 11, 16],
9 ...     ['Sally', 15, 15],

```

```

10 ...      ['Dan',16,15],
11 ...      ['Sally',13,14]])
12 >>> results = Grouping.summarize_by_value(table, 'col000',
13 ...      sum="sum(group['col001'])",
14 ...      correlation="stats.spearmanr(list(group['col001']), list(['col002']))")
15 >>> results.view()
16 +-----+-----+-----+
17 | Value | sum | correlation |
18 +-----+-----+-----+
19 | Dan   | 48  | (0.55000000000000004, 0.450000000000651308) |
20 | Sally | 52  | (0.84999999999999998, 0.1499999999962324) |
21 +-----+-----+-----+
22 >>> # the first number in the correlation is the statistic, second number is the p-value

```

# Chapter 13

## os

OS routines for Mac, NT, or Posix depending on what system we're on.

This exports: - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc. - os.path is one of the modules posixpath, or ntpath - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos' - os.curdir is a string representing the current directory ('.' or '..') - os.pardir is a string representing the parent directory ('..' or '..:') - os.sep is the (or a most common) pathname separator ('/' or ':' or '

') - os.extsep is the extension separator ('.' or '/') - os.altsep is the alternate pathname separator (None or '/') - os.pathsep is the component separator used in \$PATH etc - os.linesep is the line separator in text files ('

r' or '

n' or '

r

n') - os.defpath is the default search path for executables - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being portable between different platforms. Of course, they must then only use functions that are defined by all platforms (e.g., unlink and opendir), and leave all pathname manipulation to os.path (e.g., split and join).

### 13.1 execl

execl(file, \*args)

Execute the executable file with argument list args, replacing the current

process.

**Signature:**

```
os.execl(<object> file)
```

- <object> file:

## 13.2 execl

```
execl(file, *args, env)
```

Execute the executable file with argument list args and environment env, replacing the current process.

**Signature:**

```
os.execl(<object> file)
```

- <object> file:

## 13.3 execlp

```
execlp(file, *args)
```

Execute the executable file (which is searched for along \$PATH) with argument list args, replacing the current process.

**Signature:**

```
os.execlp(<object> file)
```

- <object> file:

## 13.4 execlpe

```
execlpe(file, *args, env)
```

Execute the executable file (which is searched for along \$PATH) with argument list args and environment env, replacing the current process.

**Signature:**



```
os.execlpe(<object> file)
```

- <object> file:

## 13.5 execvp

```
execvp(file, args)
```

Execute the executable file (which is searched for along \$PATH) with argument list args, replacing the current process. args may be a list or tuple of strings.

**Signature:**

```
os.execvp(<object> file,  
          <object> args)
```

- <object> file:
- <object> args:

## 13.6 execvpe

```
execvpe(file, args, env)
```

Execute the executable file (which is searched for along \$PATH) with argument list args and environment env , replacing the current process. args may be a list or tuple of strings.

**Signature:**

```
os.execvpe(<object> file,  
           <object> args,  
           <object> env)
```

- <object> file:
- <object> args:
- <object> env:

## 13.7 getenv

Get an environment variable, return None if it doesn't exist. The optional second argument can specify an alternate default.

**Signature:**

```
os.getenv(<object> key,  
          <object> default=None)
```

- **<object> key:**
- **<object> default (optional):** If omitted, defaults to None.

## 13.8 makedirs

```
makedirs(path [, mode=0777])
```

Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. This is recursive.

**Signature:**

```
os.makedirs(<object> name,  
            <object> mode=511)
```

- **<object> name:**
- **<object> mode (optional):** If omitted, defaults to 511.

## 13.9 popen2

Execute the shell command 'cmd' in a sub-process. On UNIX, 'cmd' may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with `os.spawnv()`). If 'cmd' is a string it will be passed to the shell (as with `os.system()`). If 'bufsize' is specified, it sets the buffer size for the I/O pipes. The file objects (`child_stdin`, `child_stdout`) are returned.

**Signature:**

```
os.popen2(<object> cmd,  
          <object> mode=t,  
          <object> bufsize=-1)
```

- **<object> cmd:**
- **<object> mode** (optional): If omitted, defaults to t.
- **<object> bufsize** (optional): If omitted, defaults to -1.

## 13.10 popen3

Execute the shell command 'cmd' in a sub-process. On UNIX, 'cmd' may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with `os.spawnv()`). If 'cmd' is a string it will be passed to the shell (as with `os.system()`). If 'bufsize' is specified, it sets the buffer size for the I/O pipes. The file objects (`child_stdin`, `child_stdout`, `child_stderr`) are returned.

### Signature:

```
os.popen3(<object> cmd,  
          <object> mode=t,  
          <object> bufsize=-1)
```

- **<object> cmd:**
- **<object> mode** (optional): If omitted, defaults to t.
- **<object> bufsize** (optional): If omitted, defaults to -1.

## 13.11 popen4

Execute the shell command 'cmd' in a sub-process. On UNIX, 'cmd' may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with `os.spawnv()`). If 'cmd' is a string it will be passed to the shell (as with `os.system()`). If 'bufsize' is specified, it sets the buffer size for the I/O pipes. The file objects (`child_stdin`, `child_stdout`, `child_stderr`) are returned.

**Signature:**

```
os.popen4(<object> cmd,  
          <object> mode=t,  
          <object> bufsize=-1)
```

- **<object> cmd:**
- **<object> mode** (optional): If omitted, defaults to t.
- **<object> bufsize** (optional): If omitted, defaults to -1.

## 13.12 removedirs

`removedirs(path)`

Super-rmdir; remove a leaf directory and all empty intermediate ones. Works like `rmdir` except that, if the leaf directory is successfully removed, directories corresponding to rightmost path segments will be pruned away until either the whole path is consumed or an error occurs. Errors during this latter phase are ignored – they generally mean that a directory was not empty.

**Signature:**

```
os.removedirs(<object> name)
```

- **<object> name:**

## 13.13 renames

`renames(old, new)`

Super-rename; create directories as necessary and delete any left empty. Works like `rename`, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away until either the whole path is consumed or a nonempty directory is found.

Note: this function can fail with the new directory structure made if you lack permissions needed to unlink the leaf directory or file.

**Signature:**

```
os.rename(<object> old,  
          <object> new)
```

- **<object> old:**
- **<object> new:**

## 13.14 spawnl

spawnl(mode, file, \*args) -> integer

Execute file with arguments from args in a subprocess. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

**Signature:**

```
os.spawnl(<object> mode,  
          <object> file)
```

- **<object> mode:**
- **<object> file:**

## 13.15 spawnle

spawnle(mode, file, \*args, env) -> integer

Execute file with arguments from args in a subprocess with the supplied environment. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

**Signature:**

```
os.spawnle(<object> mode,  
           <object> file)
```

- **<object> mode:**
- **<object> file:**

## 13.16 spawnlp

spawnlp(mode, file, \*args) -> integer

Execute file (which is looked for along \$PATH) with arguments from args in a subprocess with the supplied environment. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnlp(<object> mode,  
           <object> file)
```

- <object> mode:
- <object> file:

## 13.17 spawnlpe

spawnlpe(mode, file, \*args, env) -> integer

Execute file (which is looked for along \$PATH) with arguments from args in a subprocess with the supplied environment. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnlpe(<object> mode,  
            <object> file)
```

- <object> mode:
- <object> file:

## 13.18 spawnv

`spawnv(mode, file, args) -> integer`

Execute file with arguments from args in a subprocess. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnv(<object> mode,  
          <object> file,  
          <object> args)
```

- **<object> mode:**
- **<object> file:**
- **<object> args:**

## 13.19 spawnve

`spawnve(mode, file, args, env) -> integer`

Execute file with arguments from args in a subprocess with the specified environment. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnve(<object> mode,  
           <object> file,  
           <object> args,  
           <object> env)
```

- **<object> mode:**
- **<object> file:**
- **<object> args:**
- **<object> env:**

## 13.20 spawnvp

spawnvp(mode, file, args) -> integer

Execute file (which is looked for along \$PATH) with arguments from args in a subprocess. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnvp(<object> mode,  
           <object> file,  
           <object> args)
```

- <object> mode:
- <object> file:
- <object> args:

## 13.21 spawnvpe

spawnvpe(mode, file, args, env) -> integer

Execute file (which is looked for along \$PATH) with arguments from args in a subprocess with the supplied environment. If mode == P\_NOWAIT return the pid of the process. If mode == P\_WAIT return the process's exit code if it exits normally; otherwise return -SIG, where SIG is the signal that killed it.

### Signature:

```
os.spawnvpe(<object> mode,  
            <object> file,  
            <object> args,  
            <object> env)
```

- <object> mode:
- <object> file:
- <object> args:
- <object> env:



## 13.22 urandom

`urandom(n) -> str`

Return a string of n random bytes suitable for cryptographic use.

**Signature:**

`os.urandom(<object> n)`

- `<object> n`:

## 13.23 walk

Directory tree generator.

For each directory in the directory tree rooted at top (including top itself, but excluding '.' and '..'), yields a 3-tuple

`dirpath, dirnames, filenames`

`dirpath` is a string, the path to the directory. `dirnames` is a list of the names of the subdirectories in `dirpath` (excluding '.' and '..'). `filenames` is a list of the names of the non-directory files in `dirpath`. Note that the names in the lists are just names, with no path components. To get a full path (which begins with top) to a file or directory in `dirpath`, do `os.path.join(dirpath, name)`.

If optional arg 'topdown' is true or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top down). If `topdown` is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom up).

When `topdown` is true, the caller can modify the `dirnames` list in-place (e.g., via `del` or slice assignment), and `walk` will only recurse into the subdirectories whose names remain in `dirnames`; this can be used to prune the search, or to impose a specific order of visiting. Modifying `dirnames` when `topdown` is false is ineffective, since the directories in `dirnames` have already been generated by the time `dirnames` itself is generated.

By default errors from the `os.listdir()` call are ignored. If optional arg 'onerror' is specified, it should be a function; it will be called with one argument, an `os.error` instance. It can report the error to continue with the walk,

or raise the exception to abort the walk. Note that the filename is available as the filename attribute of the exception object.

By default, `os.walk` does not follow symbolic links to subdirectories on systems that support them. In order to get this functionality, set the optional argument `'followlinks'` to true.

Caution: if you pass a relative pathname for `top`, don't change the current working directory between resumptions of `walk`. `walk` never changes the current directory, and assumes that the client doesn't either.

### Signature:

```
os.walk(<object> top,  
        <object> topdown=True,  
        <object> onerror=None,  
        <object> followlinks=False)
```

- **<object> top:**
- **<object> topdown** (optional): If omitted, defaults to True.
- **<object> onerror** (optional): If omitted, defaults to None.
- **<object> followlinks** (optional): If omitted, defaults to False.

### Example:

```
1  
2 import os  
3 from os.path import join, getsize  
4 for root, dirs, files in os.walk('python/Lib/email'):  
5     print root, "consumes",  
6     print sum([getsize(join(root, name)) for name in files]),  
7     print "bytes in", len(files), "non-directory files"  
8     if 'CVS' in dirs:  
9         dirs.remove('CVS') # don't visit CVS directories
```

# Chapter 14

## os.path

### 14.1 abspath

Return an absolute path.

**Signature:**

```
os.path.abspath(<object> path)
```

- <object> path:

### 14.2 basename

Returns the final component of a pathname

**Signature:**

```
os.path.basename(<object> p)
```

- <object> p:

### 14.3 commonprefix

Given a list of pathnames, returns the longest common leading component

**Signature:**

`os.path.commonprefix(<object> m)`

- `<object> m:`

## 14.4 `dirname`

Returns the directory component of a pathname

**Signature:**

`os.path.dirname(<object> p)`

- `<object> p:`

## 14.5 `exists`

Test whether a path exists. Returns False for broken symbolic links

**Signature:**

`os.path.exists(<object> path)`

- `<object> path:`

## 14.6 `expanduser`

Expand `~` and `~user` constructions. If `user` or `$HOME` is unknown, do nothing.

**Signature:**

`os.path.expanduser(<object> path)`

- `<object> path:`

## 14.7 expandvars

Expand shell variables of form `$var` and `${var}`. Unknown variables are left unchanged.

**Signature:**

```
os.path.expandvars(<object> path)
```

- **<object> path:**

## 14.8 getatime

Return the last access time of a file, reported by `os.stat()`.

**Signature:**

```
os.path.getatime(<object> filename)
```

- **<object> filename:**

## 14.9 getctime

Return the metadata change time of a file, reported by `os.stat()`.

**Signature:**

```
os.path.getctime(<object> filename)
```

- **<object> filename:**

## 14.10 getmtime

Return the last modification time of a file, reported by `os.stat()`.

**Signature:**

```
os.path.getmtime(<object> filename)
```

- **<object> filename:**

## 14.11 `getsize`

Return the size of a file, reported by `os.stat()`.

**Signature:**

```
os.path.getsize(<object> filename)
```

- **<object> filename:**

## 14.12 `isabs`

Test whether a path is absolute

**Signature:**

```
os.path.isabs(<object> s)
```

- **<object> s:**

## 14.13 `isdir`

Return true if the pathname refers to an existing directory.

**Signature:**

```
os.path.isdir(<object> s)
```

- **<object> s:**

## 14.14 `isfile`

Test whether a path is a regular file

**Signature:**

```
os.path.isfile(<object> path)
```

- **<object> path:**

## 14.15 islink

Test whether a path is a symbolic link

**Signature:**

```
os.path.islink(<object> path)
```

- <object> path:

## 14.16 ismount

Test whether a path is a mount point

**Signature:**

```
os.path.ismount(<object> path)
```

- <object> path:

## 14.17 join

Join two or more pathname components, inserting '/' as needed. If any component is an absolute path, all previous path components will be discarded.

**Signature:**

```
os.path.join(<object> a)
```

- <object> a:

## 14.18 lexists

Test whether a path exists. Returns True for broken symbolic links

**Signature:**

```
os.path.lexists(<object> path)
```

- <object> path:

## 14.19 normcase

Normalize case of pathname. Has no effect under Posix

**Signature:**

```
os.path.normcase(<object> s)
```

- **<object> s:**

## 14.20 normpath

Normalize path, eliminating double slashes, etc.

**Signature:**

```
os.path.normpath(<object> path)
```

- **<object> path:**

## 14.21 realpath

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.

**Signature:**

```
os.path.realpath(<object> filename)
```

- **<object> filename:**

## 14.22 relpath

Return a relative version of a path

**Signature:**

```
os.path.relpath(<object> path,  
                <object> start=.)
```

- **<object> path:**
- **<object> start (optional):** If omitted, defaults to ..



## 14.23 samefile

Test whether two pathnames reference the same actual file

**Signature:**

```
os.path.samefile(<object> f1,  
                 <object> f2)
```

- <object> f1:
- <object> f2:

## 14.24 sameopenfile

Test whether two open file objects reference the same file

**Signature:**

```
os.path.sameopenfile(<object> fp1,  
                     <object> fp2)
```

- <object> fp1:
- <object> fp2:

## 14.25 samestat

Test whether two stat buffers reference the same file

**Signature:**

```
os.path.samestat(<object> s1,  
                 <object> s2)
```

- <object> s1:
- <object> s2:

## 14.26 split

Split a pathname. Returns tuple "(head, tail)" where "tail" is everything after the final slash. Either part may be empty.

**Signature:**

```
os.path.split(<object> p)
```

- <object> p:

## 14.27 splitdrive

Split a pathname into drive and path. On Posix, drive is always empty.

**Signature:**

```
os.path.splitdrive(<object> p)
```

- <object> p:

## 14.28 splitext

Split the extension from a pathname.

Extension is everything from the last dot to the end, ignoring leading dots. Returns "(root, ext)"; ext may be empty.

**Signature:**

```
os.path.splitext(<object> p)
```

- <object> p:

## 14.29 walk

Directory tree walk with callback function.

For each directory in the directory tree rooted at top (including top itself, but excluding '.' and '..'), call func(arg, dirname, fnames). dirname is the name of the directory, and fnames a list of the names of the files and subdirectories in dirname (excluding '.' and '..'). func may modify the fnames list in-place (e.g. via del or slice assignment), and walk will only recurse into the subdirectories whose names remain in fnames; this can be used to implement a filter, or to impose a specific order of visiting. No semantics are defined for, or required of, arg, beyond that arg is always passed to func. It can be used, e.g., to pass a filename pattern, or a mutable object designed to accumulate statistics. Passing None for arg is common.

### Signature:

```
os.path.walk(<object> top,  
             <object> func,  
             <object> arg)
```

- <object> top:
- <object> func:
- <object> arg:

# Chapter 15

## random

Random variable generators.

integers ——— uniform within range

sequences ——— pick random element pick random sample generate random permutation

distributions on the real line: ————— uniform triangular normal (Gaussian) lognormal negative exponential gamma beta pareto Weibull

distributions on the circle (angles 0 to 2pi) —————  
—— circular uniform von Mises

General notes on the underlying Mersenne Twister core generator:

\* The period is  $2^{19937}-1$ . \* It is one of the most extensively tested generators in existence. \* Without a direct way to compute N steps forward, the semantics of `jumpahead(n)` are weakened to simply jump to another distant state and rely on the large period to avoid overlapping sequences. \* The `random()` method is implemented in C, executes in a single Python step, and is, therefore, threadsafe.

### 15.1 Random

Initialize an instance.

Optional argument `x` controls seeding, as for `Random.seed()`.

**Signature:**

```
random.Random(<object> x=None)
```

- `<object> x` (optional): If omitted, defaults to `None`.

## 15.2 `Random.betavariate`

Beta distribution.

Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ . Returned values range between 0 and 1.

**Signature:**

```
[Random].betavariate(<object> alpha,  
                    <object> beta)
```

- `<object> alpha`:
- `<object> beta`:

## 15.3 `Random.choice`

Choose a random element from a non-empty sequence.

**Signature:**

```
[Random].choice(<object> seq)
```

- `<object> seq`:

## 15.4 `Random.expovariate`

Exponential distribution.

`lambda` is 1.0 divided by the desired mean. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

**Signature:**

```
[Random].expovariate(<object> lambda)
```

- `<object> lambda`:

## 15.5 Random.gammavariate

Gamma distribution. Not the gamma function!

Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ .

**Signature:**

```
[Random].gammavariate(<object> alpha,  
                      <object> beta)
```

- **<object> alpha:**
- **<object> beta:**

## 15.6 Random.gauss

Gaussian distribution.

$\mu$  is the mean, and  $\sigma$  is the standard deviation. This is slightly faster than the `normalvariate()` function.

Not thread-safe without a lock around calls.

**Signature:**

```
[Random].gauss(<object> mu,  
              <object> sigma)
```

- **<object> mu:**
- **<object> sigma:**

## 15.7 Random.getstate

Return internal state; can be passed to `setstate()` later.

**Signature:**

```
[Random].getstate()
```

## 15.8 Random.lognormvariate

Log normal distribution.

If you take the natural logarithm of this distribution, you'll get a normal distribution with mean  $\mu$  and standard deviation  $\sigma$ .  $\mu$  can have any value, and  $\sigma$  must be greater than zero.

**Signature:**

```
[Random].lognormvariate(<object> mu,  
                        <object> sigma)
```

- **<object> mu:**
- **<object> sigma:**

## 15.9 Random.normalvariate

Normal distribution.

$\mu$  is the mean, and  $\sigma$  is the standard deviation.

**Signature:**

```
[Random].normalvariate(<object> mu,  
                       <object> sigma)
```

- **<object> mu:**
- **<object> sigma:**

## 15.10 Random.paretovariate

Pareto distribution.  $\alpha$  is the shape parameter.

**Signature:**

```
[Random].paretovariate(<object> alpha)
```

- **<object> alpha:**

## 15.11 Random.randint

Return random integer in range [a, b], including both end points.

**Signature:**

```
[Random].randint(<object> a,  
                 <object> b)
```

- <object> a:
- <object> b:

## 15.12 Random.randrange

Choose a random item from range(start, stop[, step]).

This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want. Do not supply the 'int', 'default', and 'maxwidth' arguments.

**Signature:**

```
[Random].randrange(<object> start,  
                  <object> stop=None,  
                  <object> step=1,  
                  <object> int=<type 'int'>,  
                  <object> default=None,  
                  <object> maxwidth=9007199254740992)
```

- <object> start:
- <object> stop (optional): If omitted, defaults to None.
- <object> step (optional): If omitted, defaults to 1.
- <object> int (optional): If omitted, defaults to <type 'int'>.
- <object> default (optional): If omitted, defaults to None.
- <object> maxwidth (optional): If omitted, defaults to 9007199254740992.



## 15.13 Random.sample

Chooses k unique random elements from a population sequence.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample in a range of integers, use xrange as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(100000000), 60)`

### Signature:

```
[Random].sample(<object> population,  
                <object> k)
```

- **<object> population:**
- **<object> k:**

## 15.14 Random.seed

Initialize internal state from hashable object.

None or no argument seeds from current time or from an operating system specific randomness source if available.

If a is not None or an int or long, hash(a) is used instead.

### Signature:

```
[Random].seed(<object> a=None)
```

- **<object> a (optional):** If omitted, defaults to None.

## 15.15 Random.setstate

Restore internal state from object returned by `getstate()`.

**Signature:**

```
[Random].setstate(<object> state)
```

- **<object> state:**

## 15.16 Random.shuffle

`x`, `random=random.random` -> shuffle list `x` in place; return `None`.

Optional arg `random` is a 0-argument function returning a random float in `[0.0, 1.0)`; by default, the standard `random.random`.

**Signature:**

```
[Random].shuffle(<object> x,  
                 <object> random=None,  
                 <object> int=<type 'int'>)
```

- **<object> x:**
- **<object> random (optional):** If omitted, defaults to `None`.
- **<object> int (optional):** If omitted, defaults to `<type 'int'>`.

## 15.17 Random.triangular

Triangular distribution.

Continuous distribution bounded by given lower and upper limits, and having a given mode value in-between.

[http://en.wikipedia.org/wiki/Triangular\\_distribution](http://en.wikipedia.org/wiki/Triangular_distribution)

**Signature:**

```
[Random].triangular(<object> low=0.0,  
                   <object> high=1.0,  
                   <object> mode=None)
```

- `<object> low` (optional): If omitted, defaults to 0.0.
- `<object> high` (optional): If omitted, defaults to 1.0.
- `<object> mode` (optional): If omitted, defaults to None.

## 15.18 Random.uniform

Get a random number in the range  $[a, b)$ .

**Signature:**

```
[Random].uniform(<object> a,  
                 <object> b)
```

- `<object> a`:
- `<object> b`:

## 15.19 Random.vonmisesvariate

Circular data distribution.

$\mu$  is the mean angle, expressed in radians between 0 and  $2\pi$ , and  $\kappa$  is the concentration parameter, which must be greater than or equal to zero. If  $\kappa$  is equal to zero, this distribution reduces to a uniform random angle over the range 0 to  $2\pi$ .

**Signature:**

```
[Random].vonmisesvariate(<object> mu,  
                        <object> kappa)
```

- `<object> mu`:
- `<object> kappa`:

## 15.20 Random.weibullvariate

Weibull distribution.

alpha is the scale parameter and beta is the shape parameter.

**Signature:**

```
[Random].weibullvariate(<object> alpha,  
                        <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.21 SystemRandom

Initialize an instance.

Optional argument x controls seeding, as for Random.seed().

**Signature:**

```
random.SystemRandom(<object> x=None)
```

- <object> x (optional): If omitted, defaults to None.

## 15.22 SystemRandom.betavariate

Beta distribution.

Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ . Returned values range between 0 and 1.

**Signature:**

```
[SystemRandom].betavariate(<object> alpha,  
                           <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.23 SystemRandom.choice

Choose a random element from a non-empty sequence.

**Signature:**

```
[SystemRandom].choice(<object> seq)
```

- <object> seq:

## 15.24 SystemRandom.expovariate

Exponential distribution.

lambd is 1.0 divided by the desired mean. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

**Signature:**

```
[SystemRandom].expovariate(<object> lambd)
```

- <object> lambd:

## 15.25 SystemRandom.gammavariate

Gamma distribution. Not the gamma function!

Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ .

**Signature:**

```
[SystemRandom].gammavariate(<object> alpha,  
                              <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.26 SystemRandom.gauss

Gaussian distribution.

mu is the mean, and sigma is the standard deviation. This is slightly faster than the `normalvariate()` function.

Not thread-safe without a lock around calls.

**Signature:**

```
[SystemRandom].gauss(<object> mu,  
                     <object> sigma)
```

- **<object> mu:**
- **<object> sigma:**

## 15.27 SystemRandom.getrandbits

`getrandbits(k) -> x`. Generates a long int with k random bits.

**Signature:**

```
[SystemRandom].getrandbits(<object> k)
```

- **<object> k:**

## 15.28 SystemRandom.getstate

Method should not be called for a system random number generator.

**Signature:**

```
[SystemRandom].getstate()
```

## 15.29 SystemRandom.jumpahead

Stub method. Not used for a system random number generator.

**Signature:**

```
[SystemRandom].jumpahead()
```

## 15.30 SystemRandom.lognormvariate

Log normal distribution.

If you take the natural logarithm of this distribution, you'll get a normal distribution with mean `mu` and standard deviation `sigma`. `mu` can have any value, and `sigma` must be greater than zero.

**Signature:**

```
[SystemRandom].lognormvariate(<object> mu,  
                               <object> sigma)
```

- `<object> mu`:
- `<object> sigma`:

## 15.31 SystemRandom.normalvariate

Normal distribution.

`mu` is the mean, and `sigma` is the standard deviation.

**Signature:**

```
[SystemRandom].normalvariate(<object> mu,  
                              <object> sigma)
```

- `<object> mu`:
- `<object> sigma`:

## 15.32 SystemRandom.paretovariate

Pareto distribution. `alpha` is the shape parameter.

**Signature:**

```
[SystemRandom].paretovariate(<object> alpha)
```

- `<object> alpha`:

### 15.33 SystemRandom.randint

Return random integer in range [a, b], including both end points.

**Signature:**

```
[SystemRandom].randint(<object> a,  
                        <object> b)
```

- <object> a:
- <object> b:

### 15.34 SystemRandom.random

Get the next random number in the range [0.0, 1.0).

**Signature:**

```
[SystemRandom].random()
```

### 15.35 SystemRandom.randrange

Choose a random item from range(start, stop[, step]).

This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want. Do not supply the 'int', 'default', and 'maxwidth' arguments.

**Signature:**

```
[SystemRandom].randrange(<object> start,  
                          <object> stop=None,  
                          <object> step=1,  
                          <object> int=<type 'int'>,  
                          <object> default=None,  
                          <object> maxwidth=9007199254740992)
```

- <object> start:
- <object> stop (optional): If omitted, defaults to None.



- `<object> step` (optional): If omitted, defaults to 1.
- `<object> int` (optional): If omitted, defaults to `<type 'int'>`.
- `<object> default` (optional): If omitted, defaults to None.
- `<object> maxwidth` (optional): If omitted, defaults to 9007199254740992.

### 15.36 SystemRandom.sample

Chooses k unique random elements from a population sequence.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample in a range of integers, use `xrange` as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`

#### Signature:

```
[SystemRandom].sample(<object> population,  
                      <object> k)
```

- `<object> population`:
- `<object> k`:

### 15.37 SystemRandom.seed

Stub method. Not used for a system random number generator.

#### Signature:

```
[SystemRandom].seed()
```

## 15.38 SystemRandom.setstate

Method should not be called for a system random number generator.

**Signature:**

```
[SystemRandom].setstate()
```

## 15.39 SystemRandom.shuffle

x, random=random.random -> shuffle list x in place; return None.

Optional arg random is a 0-argument function returning a random float in [0.0, 1.0); by default, the standard random.random.

**Signature:**

```
[SystemRandom].shuffle(<object> x,  
                        <object> random=None,  
                        <object> int=<type 'int'>)
```

- **<object> x:**
- **<object> random (optional):** If omitted, defaults to None.
- **<object> int (optional):** If omitted, defaults to <type 'int'>.

## 15.40 SystemRandom.triangular

Triangular distribution.

Continuous distribution bounded by given lower and upper limits, and having a given mode value in-between.

[http://en.wikipedia.org/wiki/Triangular\\_distribution](http://en.wikipedia.org/wiki/Triangular_distribution)

**Signature:**

```
[SystemRandom].triangular(<object> low=0.0,  
                           <object> high=1.0,  
                           <object> mode=None)
```

- `<object> low` (optional): If omitted, defaults to 0.0.
- `<object> high` (optional): If omitted, defaults to 1.0.
- `<object> mode` (optional): If omitted, defaults to None.

## 15.41 `SystemRandom.uniform`

Get a random number in the range `[a, b)`.

**Signature:**

```
[SystemRandom].uniform(<object> a,  
                        <object> b)
```

- `<object> a`:
- `<object> b`:

## 15.42 `SystemRandom.vonmisesvariate`

Circular data distribution.

`mu` is the mean angle, expressed in radians between 0 and  $2\pi$ , and `kappa` is the concentration parameter, which must be greater than or equal to zero. If `kappa` is equal to zero, this distribution reduces to a uniform random angle over the range 0 to  $2\pi$ .

**Signature:**

```
[SystemRandom].vonmisesvariate(<object> mu,  
                               <object> kappa)
```

- `<object> mu`:
- `<object> kappa`:

## 15.43 SystemRandom.weibullvariate

Weibull distribution.

alpha is the scale parameter and beta is the shape parameter.

**Signature:**

```
[SystemRandom].weibullvariate(<object> alpha,  
                               <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.44 WichmannHill

Initialize an instance.

Optional argument x controls seeding, as for Random.seed().

**Signature:**

```
random.WichmannHill(<object> x=None)
```

- <object> x (optional): If omitted, defaults to None.

## 15.45 WichmannHill.betavariate

Beta distribution.

Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ . Returned values range between 0 and 1.

**Signature:**

```
[WichmannHill].betavariate(<object> alpha,  
                            <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.46 WichmannHill.choice

Choose a random element from a non-empty sequence.

**Signature:**

```
[WichmannHill].choice(<object> seq)
```

- <object> seq:

## 15.47 WichmannHill.expovariate

Exponential distribution.

    lamdb is 1.0 divided by the desired mean. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

**Signature:**

```
[WichmannHill].expovariate(<object> lamdb)
```

- <object> lamdb:

## 15.48 WichmannHill.gammavariate

Gamma distribution. Not the gamma function!

    Conditions on the parameters are  $\alpha > 0$  and  $\beta > 0$ .

**Signature:**

```
[WichmannHill].gammavariate(<object> alpha,  
                             <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.49 WichmannHill.gauss

Gaussian distribution.

mu is the mean, and sigma is the standard deviation. This is slightly faster than the `normalvariate()` function.

Not thread-safe without a lock around calls.

**Signature:**

```
[WichmannHill].gauss(<object> mu,  
                    <object> sigma)
```

- **<object> mu:**
- **<object> sigma:**

## 15.50 WichmannHill.getstate

Return internal state; can be passed to `setstate()` later.

**Signature:**

```
[WichmannHill].getstate()
```

## 15.51 WichmannHill.jumpahead

Act as if `n` calls to `random()` were made, but quickly.

`n` is an int, greater than or equal to 0.

Example use: If you have 2 threads and know that each will consume no more than a million random numbers, create two Random objects `r1` and `r2`, then do `r2.setstate(r1.getstate())` `r2.jumpahead(1000000)` Then `r1` and `r2` will use guaranteed-disjoint segments of the full period.

**Signature:**

```
[WichmannHill].jumpahead(<object> n)
```

- **<object> n:**

## 15.52 WichmannHill.lognormvariate

Log normal distribution.

If you take the natural logarithm of this distribution, you'll get a normal distribution with mean `mu` and standard deviation `sigma`. `mu` can have any value, and `sigma` must be greater than zero.

**Signature:**

```
[WichmannHill].lognormvariate(<object> mu,  
                               <object> sigma)
```

- `<object> mu`:
- `<object> sigma`:

## 15.53 WichmannHill.normalvariate

Normal distribution.

`mu` is the mean, and `sigma` is the standard deviation.

**Signature:**

```
[WichmannHill].normalvariate(<object> mu,  
                              <object> sigma)
```

- `<object> mu`:
- `<object> sigma`:

## 15.54 WichmannHill.paretovariate

Pareto distribution. `alpha` is the shape parameter.

**Signature:**

```
[WichmannHill].paretovariate(<object> alpha)
```

- `<object> alpha`:

## 15.55 WichmannHill.randint

Return random integer in range [a, b], including both end points.

**Signature:**

```
[WichmannHill].randint(<object> a,  
                        <object> b)
```

- <object> a:
- <object> b:

## 15.56 WichmannHill.random

Get the next random number in the range [0.0, 1.0).

**Signature:**

```
[WichmannHill].random()
```

## 15.57 WichmannHill.randrange

Choose a random item from range(start, stop[, step]).

This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want. Do not supply the 'int', 'default', and 'maxwidth' arguments.

**Signature:**

```
[WichmannHill].randrange(<object> start,  
                        <object> stop=None,  
                        <object> step=1,  
                        <object> int=<type 'int'>,  
                        <object> default=None,  
                        <object> maxwidth=9007199254740992)
```

- <object> start:
- <object> stop (optional): If omitted, defaults to None.



- `<object> step` (optional): If omitted, defaults to 1.
- `<object> int` (optional): If omitted, defaults to `<type 'int'>`.
- `<object> default` (optional): If omitted, defaults to None.
- `<object> maxwidth` (optional): If omitted, defaults to 9007199254740992.

## 15.58 WichmannHill.sample

Chooses `k` unique random elements from a population sequence.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample in a range of integers, use `xrange` as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`

### Signature:

```
[WichmannHill].sample(<object> population,  
                      <object> k)
```

- `<object> population`:
- `<object> k`:

## 15.59 WichmannHill.seed

Initialize internal state from hashable object.

None or no argument seeds from current time or from an operating system specific randomness source if available.

If `a` is not None or an int or long, `hash(a)` is used instead.

If `a` is an `int` or `long`, `a` is used directly. Distinct values between 0 and 27814431486575L inclusive are guaranteed to yield distinct internal states (this guarantee is specific to the default Wichmann-Hill generator).

**Signature:**

```
[WichmannHill].seed(<object> a=None)
```

- `<object> a` (optional): If omitted, defaults to `None`.

## 15.60 WichmannHill.setstate

Restore internal state from object returned by `getstate()`.

**Signature:**

```
[WichmannHill].setstate(<object> state)
```

- `<object> state`:

## 15.61 WichmannHill.shuffle

`x, random=random.random -> shuffle list x in place; return None.`

Optional arg `random` is a 0-argument function returning a random float in `[0.0, 1.0)`; by default, the standard `random.random`.

**Signature:**

```
[WichmannHill].shuffle(<object> x,  
                        <object> random=None,  
                        <object> int=<type 'int'>)
```

- `<object> x`:
- `<object> random` (optional): If omitted, defaults to `None`.
- `<object> int` (optional): If omitted, defaults to `<type 'int'>`.

## 15.62 WichmannHill.triangular

Triangular distribution.

Continuous distribution bounded by given lower and upper limits, and having a given mode value in-between.

[http://en.wikipedia.org/wiki/Triangular\\_distribution](http://en.wikipedia.org/wiki/Triangular_distribution)

**Signature:**

```
[WichmannHill].triangular(<object> low=0.0,  
                           <object> high=1.0,  
                           <object> mode=None)
```

- <object> low (optional): If omitted, defaults to 0.0.
- <object> high (optional): If omitted, defaults to 1.0.
- <object> mode (optional): If omitted, defaults to None.

## 15.63 WichmannHill.uniform

Get a random number in the range [a, b).

**Signature:**

```
[WichmannHill].uniform(<object> a,  
                       <object> b)
```

- <object> a:
- <object> b:

## 15.64 WichmannHill.vonmisesvariate

Circular data distribution.

$\mu$  is the mean angle, expressed in radians between 0 and  $2\pi$ , and  $\kappa$  is the concentration parameter, which must be greater than or equal to zero. If  $\kappa$  is equal to zero, this distribution reduces to a uniform random angle over the range 0 to  $2\pi$ .

**Signature:**

```
[WichmannHill].vonmisesvariate(<object> mu,  
                                <object> kappa)
```

- <object> mu:
- <object> kappa:

## 15.65 WichmannHill.weibullvariate

Weibull distribution.

alpha is the scale parameter and beta is the shape parameter.

**Signature:**

```
[WichmannHill].weibullvariate(<object> alpha,  
                               <object> beta)
```

- <object> alpha:
- <object> beta:

## 15.66 WichmannHill.whseed

Seed from hashable object's hash code.

None or no argument seeds from current time. It is not guaranteed that objects with distinct hash codes lead to distinct internal states.

This is obsolete, provided for compatibility with the seed routine used prior to Python 2.1. Use the .seed() method instead.

**Signature:**

```
[WichmannHill].whseed(<object> a=None)
```

- <object> a (optional): If omitted, defaults to None.

# Chapter 16

## re

Support for regular expressions (RE).

This module provides regular expression matching operations similar to those found in Perl. It supports both 8-bit and Unicode strings; both the pattern and the strings being processed can contain null bytes and characters outside the US ASCII range.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like "A", "a", or "0", are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string 'last'.

The special characters are: `.` Matches any character except a newline. `^` Matches the start of the string. `$` Matches the end of the string or just before the newline at the end of the string. `*` Matches 0 or more (greedy) repetitions of the preceding RE. Greedy means that it will match as many repetitions as possible. `+` Matches 1 or more (greedy) repetitions of the preceding RE. `?` Matches 0 or 1 (greedy) of the preceding RE. `*?+?+?` Non-greedy versions of the previous three special characters. `{m,n}` Matches from m to n repetitions of the preceding RE. `{m,n}?` Non-greedy version of the above. `"`

`"` Either escapes special characters or signals a special sequence. `[]` Indicates a set of characters. A `"^"` as the first character indicates a complementing set. `"—"` A—B, creates an RE that will match either A or B. `(...)` Matches the RE inside the parentheses. The contents can be retrieved or matched later in the string. `(?iLmsux)` Set the I, L, M, S, U, or X flag for the RE (see below). `(?:...)` Non-grouping version of regular parentheses. `(?P<name>...)` The

substring matched by the group is accessible by name. (`?P=name`) Matches the text matched earlier by the group named name. (`?#...`) A comment; ignored. (`?=...`) Matches if ... matches next, but doesn't consume the string. (`?!...`) Matches if ... doesn't match next. (`?<=...`) Matches if preceded by ... (must be fixed length). (`?<!...`) Matches if not preceded by ... (must be fixed length). (`?(id/name)yes—no`) Matches yes pattern if the group with id/name matched, the (optional) no pattern otherwise.

The special sequences consist of "

" and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character.

number Matches the contents of the group of the same number.

A Matches only at the start of the string.

Z Matches only at the end of the string.

b Matches the empty string, but only at the start or end of a word.

B Matches the empty string, but not at the start or end of a word.

d Matches any decimal digit; equivalent to the set `[0-9]`.

D Matches any non-digit character; equivalent to the set `[^0-9]`.

s Matches any whitespace character; equivalent to `[`

t

n

r

f

v].

S Matches any non-whitespace character; equiv. to `[^`

t

n

r

f

v].

w Matches any alphanumeric character; equivalent to `[a-zA-Z0-9_]`. With `LOCALE`, it will match the set `[0-9_]` plus characters defined as letters for the current locale.

W Matches the complement of

w.

Matches a literal backslash.

This module exports the following functions: `match` Match a regular

expression pattern to the beginning of a string. `search` Search a string for the presence of a pattern. `sub` Substitute occurrences of a pattern found in a string. `subn` Same as `sub`, but also return the number of substitutions made. `split` Split a string by the occurrences of a pattern. `findall` Find all occurrences of a pattern in a string. `finditer` Return an iterator yielding a match object for each match. `compile` Compile a pattern into a `RegexObject`. `purge` Clear the regular expression cache. `escape` Backslash all non-alphanumerics in a string.

Some of the functions in this module takes flags as optional parameters: I IGNORECASE Perform case-insensitive matching. L LOCALE Make

w,

W,

b,

B, dependent on the current locale. M MULTILINE `""^` matches the beginning of lines (after a newline) as well as the string. `""$` matches the end of lines (before a newline) as well as the end of the string. S DOTALL `"".` matches any character at all, including the newline. X VERBOSE Ignore whitespace and comments for nicer looking RE's. U UNICODE Make

w,

W,

b,

B, dependent on the Unicode locale.

This module also defines an exception `'error'`.

## 16.1 compile

Compile a regular expression pattern, returning a pattern object.

### Signature:

```
re.compile(<object> pattern,
           <object> flags=0)
```

- **<object> pattern:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.2 escape

Escape all non-alphanumeric characters in pattern.

**Signature:**

```
re.escape(<object> pattern)
```

- **<object> pattern:**

## 16.3 findall

Return a list of all non-overlapping matches in the string.

If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.

Empty matches are included in the result.

**Signature:**

```
re.findall(<object> pattern,  
          <object> string,  
          <object> flags=0)
```

- **<object> pattern:**
- **<object> string:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.4 finditer

Return an iterator over all non-overlapping matches in the string. For each match, the iterator returns a match object.

Empty matches are included in the result.

**Signature:**

```
re.finditer(<object> pattern,  
           <object> string,  
           <object> flags=0)
```



- **<object> pattern:**
- **<object> string:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.5 match

Try to apply the pattern at the start of the string, returning a match object, or None if no match was found.

**Signature:**

```
re.match(<object> pattern,  
         <object> string,  
         <object> flags=0)
```

- **<object> pattern:**
- **<object> string:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.6 purge

Clear the regular expression cache

**Signature:**

```
re.purge()
```

## 16.7 Scanner

None

**Signature:**

```
re.Scanner(<object> lexicon,  
          <object> flags=0)
```

- **<object> lexicon:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.8 Scanner.scan

None

**Signature:**

```
[Scanner].scan(<object> string)
```

- **<object> string:**

## 16.9 search

Scan through string looking for a match to the pattern, returning a match object, or None if no match was found.

**Signature:**

```
re.search(<object> pattern,  
          <object> string,  
          <object> flags=0)
```

- **<object> pattern:**
- **<object> string:**
- **<object> flags (optional):** If omitted, defaults to 0.

## 16.10 split

Split the source string by the occurrences of the pattern, returning a list containing the resulting substrings.

**Signature:**

```
re.split(<object> pattern,  
        <object> string,  
        <object> maxsplit=0)
```

- **<object> pattern:**
- **<object> string:**
- **<object> maxsplit (optional):** If omitted, defaults to 0.

## 16.11 sub

Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. repl can be either a string or a callable; if a callable, it's passed the match object and must return a replacement string to be used.

**Signature:**

```
re.sub(<object> pattern,  
      <object> repl,  
      <object> string,  
      <object> count=0)
```

- **<object> pattern:**
- **<object> repl:**
- **<object> string:**
- **<object> count (optional):** If omitted, defaults to 0.

## 16.12 subn

Return a 2-tuple containing (new\_string, number). new\_string is the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in the source string by the replacement repl. number is the number of substitutions that were made. repl can be either a string or a callable; if a

callable, it's passed the match object and must return a replacement string to be used.

**Signature:**

```
re.subn(<object> pattern,  
        <object> repl,  
        <object> string,  
        <object> count=0)
```

- **<object> pattern:**
- **<object> repl:**
- **<object> string:**
- **<object> count (optional):** If omitted, defaults to 0.

## 16.13 template

Compile a template pattern, returning a pattern object

**Signature:**

```
re.template(<object> pattern,  
            <object> flags=0)
```

- **<object> pattern:**
- **<object> flags (optional):** If omitted, defaults to 0.

# Chapter 17

## Simple

The Simple module contains base functions that are useful throughout the Picalo package, such as indexing, sorting, selecting, and matching of tables.

### 17.1 col\_join

Joins two tables together (similar to a regular SQL inner join) that have matching values in the given columns.

This function is much faster than `Simple.join()` because it uses table indices. However, it is much slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

#### Signature:

```
<Table> = Simple.col_join(<Table> table1,  
                          <Table> table2)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

Returns: A picalo Table containing matching records from the two source tables

#### Example:

```

1 >>> table1 = Table([( 'col000', int), ( 'col001', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000', int), ( 'col001', unicode), ( 'col002', int)], ([3, 'Bailey', 2], [1, 'D
3 >>> matches = Simple.col_join(table1, table2, [ 'col000', 'col000'], [ 'col001', 'col002'])
4 >>> matches.view()
5 +-----+-----+-----+-----+-----+
6 | col000 | col001 | col000_2 | col001_2 | col002 |
7 +-----+-----+-----+-----+-----+
8 |      3 |      2 |      3 | Bailey |      2 |
9 |      3 |      2 |      3 | Sally  |      2 |
10 +-----+-----+-----+-----+-----+

```

## 17.2 col\_left\_join

Joins two tables together (similar to a regular SQL LEFT join) that have matching values in the given columns. All records in table1 are returned, and only matching records in table2 are joined with them.

This function is much faster than Simple.join() because it uses table indices. However, it is much slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

### Signature:

```

<Table> = Simple.col_left_join(<Table> table1,
                              <Table> table2)

```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

Returns: A picalo Table containing matching records from the two source tables

### Example:

```

1 >>> table1 = Table([( 'col000', int), ( 'col001', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000', int), ( 'col001', unicode), ( 'col002', int)], ([3, 'Bailey', 2], [1, 'D
3 >>> matches = Simple.col_left_join(table1, table2, [ 'col000', 'col000'], [ 'col001', 'col002'])
4 >>> matches.view()
5 +-----+-----+-----+-----+-----+
6 | col000 | col001 | col000_1 | col001_1 | col002 |
7 +-----+-----+-----+-----+-----+
8 |      3 |      2 |      3 | Bailey |      2 |
9 |      3 |      2 |      3 | Sally  |      2 |
10 |      4 |      5 |    <N> |    <N> |    <N> |
11 +-----+-----+-----+-----+-----+

```

## 17.3 col\_match

Finds rows in the two tables with values that match in specific columns. This method is usually used only internally, but it is available for general use for those who want it. (in other words, most users generally use other methods like join, col\_match\_same, etc.)

### Signature:

```
<Table> = Simple.col_match(<Table> table1,
                           <Table> table2)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

Returns: A picalo Table containing the table1 match, the table2 match, and the key

### Example:

```
1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ', 2], [1, 'Da
3 >>> matches = Simple.col_match(table1, table2, [ 'col000 ', 'col000 '], [ 'col001 ', 'col002 '])
4 >>> matches.view()
5 +-----+-----+-----+
6 | table1record | table2record | key |
7 +-----+-----+-----+
8 |           0 |           0 | (3, 2) |
9 |           0 |           2 | (3, 2) |
10 +-----+-----+-----+
```

## 17.4 col\_match\_diff

Finds records in the two tables with values that match in specific columns and returns two new tables that contain everything but those records. Stated differently, this method filters all matching rows out of the two tables and returns the filtered tables (the original tables are not modified). All records that do not have matching keys in the other table are included.

This function is the opposite of col\_match\_same.

### Signature:

```
<list> = Simple.col_match_diff(<Table> table1,
                               <Table> table2)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

Returns: A list of two Tables containing only non-matching records

### Example:

```
1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ', 2], [1, 'Da
3 >>> match1, match2 = Simple.col_match_diff(table1, table2, [ 'col000 ', 'col000 '])
4 >>> match1.view()
5 +-----+
6 | col000 | col001 |
7 +-----+
8 |      4 |      5 |
9 +-----+
10 >>> match2.view()
11 +-----+-----+
12 | col000 | col001 | col002 |
13 +-----+-----+
14 |      1 |  Dan  |      2 |
15 +-----+-----+
```

## 17.5 col\_match\_same

Finds records in the two tables with values that match in specific columns and returns two new tables that contain only those records. Stated differently, this method filters all non-matching rows out of the two tables and returns the filtered tables (the original tables are not modified). All records that have matching keys in the other table are included.

This function is the opposite of `col_match_diff`.

### Signature:

```
<list> = Simple.col_match_same(<Table> table1,
                               <Table> table2)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table



Returns: A list of two Tables containing only matching records

**Example:**

```

1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ',2], [1, 'D
3 >>> match1, match2 = Simple.col_match_same(table1, table2, [ 'col000 ', 'col000 '], [ 'col001 ', 'col002 '])
4 >>> match1.view()
5 +-----+-----+
6 | col000 | col001 |
7 +-----+-----+
8 |      3 |      2 |
9 +-----+-----+
10 >>> match2.view()
11 +-----+-----+-----+
12 | col000 | col001 | col002 |
13 +-----+-----+-----+
14 |      3 | Bailey |      2 |
15 |      3 | Sally  |      2 |
16 +-----+-----+-----+

```

## 17.6 col\_right\_join

Joins two tables together (similar to a regular SQL RIGHT join) that have matching values in the given columns. All records in the table2 are returned, and only matching records in table1 are joined with them.

This function is much faster than Simple.join() because it uses table indices. However, it is much slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

**Signature:**

```

<Table> = Simple.col_right_join(<Table> table1,
                                <Table> table2)

```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

Returns: A picalo Table containing matching records from the two source tables

**Example:**

```

1 >>> table1 = Table([('col000', int), ('col001', int)], ([3,2], [4,5]))
2 >>> table2 = Table([('col000', int), ('col001', unicode), ('col002', int)], ([3,'Bailey',2], [1,'D
3 >>> matches = Simple.col_right_join(table1, table2, ['col000','col000'], ['col001','col002'])
4 >>> matches.view()
5 +-----+-----+-----+-----+-----+
6 | col000 | col001 | col002 | col000_1 | col001_1 |
7 +-----+-----+-----+-----+-----+
8 |      3 | Bailey |      2 |      3   |      2   |
9 |      1 | Dan   |      2 |    <N>   |    <N>   |
10 |      3 | Sally |      2 |      3   |      2   |
11 +-----+-----+-----+-----+-----+

```

## 17.7 compare\_records

Runs through a table sequentially and compares each record with the one after it. In other words, if table has 4 records, this method compares  $0 \leq 1$ ,  $1 \leq 2$ , and  $2 \leq 3$ . It runs the given expression for each set and returns the indices of those sets that return True. The expression should always evaluate to True or False.

The expression should evaluate record1 against record2, as in: `{ "record1['id'] == record2['id'] - 1" }`

The index of the first record is stored in the results list. So if the expression compares the third and fourth records and evaluates true, index 3 is stored in the table.

### Signature:

```
<Table> = Simple.compare_records(<Table or TableArray> table,
                                <str> expression)
```

- **<Table or TableArray> table:** The table to be analyzed
- **<str> expression:** The expression to compare each record set with

Returns: A single-column table of indices that the function returned true for

### Example:

```

1 >>> table = Table([('col000', int)], ([8000], [2000], [9000], [10000], [8000]))
2 >>> results = Simple.compare_records(table, "record1.col000 < record2.col000")
3 >>> results.view()
4 +-----+
5 | Row Indices |

```

6	+	+
7		1
8		2
9	+	+

```

10
11 In the above example, the following comparisons are made:
12 8000 < 2000 (false)
13 2000 < 9000 (true)
14 9000 < 10000 (true)
15 10000 < 8000 (false)

```

## 17.8 custom\_match

Finds rows in the two tables with values that match as returned by the specified expression. This is a more general version of `col_match`. It is significantly slower than `col_match` because it can't use indices. It is  $O^2$ .

The expression should use "record1" for the current record in table 1 and "record2" for the current record in the table 2, and it should evaluate to True or False.

### Signature:

```

<Table> = Simple.custom_match(<Table> table1,
                              <Table> table2,
                              <str> expression)

```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table
- **<str> expression:** An expression to use for the match

Returns: A picalo Table containing the table1 match and the table2 match

### Example:

```

1 >>> table1 = Table([('col000', int), ('col001', int)], ([3,2], [4,5]))
2 >>> table2 = Table([('col000', int), ('col001', unicode), ('col002', int)], ([3,'Bailey',2], [1,'De
3 >>> # match if the first column matches (granted, a very simple comparison in this example)P
4 >>> matches = Simple.custom_match(table1, table2, "record1[0] == record2[0]")
5 >>> matches.view()
6 +-----+
7 | table1record | table2record |
8 +-----+

```

9			0			0	
10			0			2	
11	+	-----	+	-----	+	+	

## 17.9 custom\_match\_diff

Finds records in the two tables with values that do not match based upon the give expression and returns two new tables that contain only those records. Stated differently, this method filters all matching rows out of the two tables and returns the filtered tables (the original tables are not modified). All records that do not have matching keys in the other table are included.

The expression should use "record1" for the current record in table 1 and "record2" for the current record in the table 2, and it should evaluate to True or False.

This function is the opposite of `custom_match_same`.

### Signature:

```
<list> = Simple.custom_match_diff(<Table> table1,
                                   <Table> table2,
                                   <str> expression)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table
- **<str> expression:** An expression to use for the match

Returns: A list of two Tables containing only non-matching records

### Example:

```
1 >>> table1 = Table([( 'col000', int), ( 'col001', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000', int), ( 'col001', unicode), ( 'col002', int)], ([3, 'Bailey', 2], [1, 'Da
3 >>> # match if the first column matches (granted, a very simple comparison in this example)
4 >>> match1, match2 = Simple.custom_match_diff(table1, table2, "record1[0] == record2[0]")
5 >>> match1.view()
6 +-----+-----+
7 | col000 | col001 |
8 +-----+-----+
9 |      3 |      2 |
10 +-----+-----+
11 >>> match2.view()
12 +-----+-----+-----+-----+
```

13		col000		col001		col002	
14							
15		3		Bailey		2	
16		3		Sally		2	
17							

## 17.10 custom\_match\_same

Finds records in the two tables with values that match based upon the give expression and returns two new tables that contain only those records. Stated differently, this method filters all non-matching rows out of the two tables and returns the filtered tables (the original tables are not modified). All records that have matching keys in the other table are included.

The expression should use "record1" for the current record in table 1 and "record2" for the current record in the table 2, and it should evaluate to True or False.

This function is the opposite of `custom_match_diff`.

### Signature:

```
<list> = Simple.custom_match_same(<Table> table1,
                                   <Table> table2,
                                   <str> expression)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table
- **<str> expression:** An expression to use for the match

Returns: list of two Tables containing only matching records

### Example:

```
1 >>> employees = Table([( 'col000 ', unicode), ( 'col001 ', unicode)], ([ 'Bailey ', '123 North Way'], [ 'S
2 >>> vendors = Table([( 'col000 ', unicode), ( 'col001 ', unicode)], ([ 'ABC Comp ', '789 Maple'], [ 'DEF E
3 >>> t1, t2 = Simple.custom_match_same(vendors, employees, "Simple.fuzzymatch(record1[1], record2[1])
4 >>> t1.view()
5 +-----+-----+
6 |           col000           |           col001           |
7 +-----+-----+
8 | DEF Enterprises | 123 Nth Way |
9 +-----+-----+
10 >>> t2.view()
```

11			
12		col000	col001
13			
14		Bailey	123 North Way
15			

## 17.11 describe

Prepares a set of statistical descriptives for the given columns in the given table. While these descriptives could be retrieved from the included stats module, this method gives quick and easy access to the main statistical measures, such as mean, median, counts, totals, and so forth. The descriptives available are shown in the example below.

If no columns are designated, descriptives are run for every column in the table. Note that this method is not optimized yet, so it might take a while for large tables with many columns.

For numeric descriptives like the mean or std deviation, text fields and empty fields are ignored. If no numeric descriptives can be calculated, the field is set to None. Since this function is meant to show a descriptive view of a table (and not to provide hard statistical values), most fields are rounded off to 2 decimal places for readability.

### Signature:

```
<Table> = Simple.describe(<Table or TableArray> table)
```

- **<Table or TableArray> table:** The table to run descriptives on.

Returns: A picalo table giving descriptives for the columns.

### Example:

```
1 >>> table = Table([
2 ...     'Name',
3 ...     'Pay',
4 ... ], [
5 ...     ['Homer', 0],
6 ...     ['Marge', 20],
7 ...     ['Bart', None],
8 ...     ['Lisa', 152],
9 ...     ['Maggie', ''],
10 ... ])
11 >>> descriptives = Simple.describe(table)
12 >>> descriptives.view()
```

13			
14	Descriptive	Name	Pay
15			
16	NumRecords	5	5
17	NumEmpty	0	2
18	NumZero	0	1
19	NumText	5	0
20	NumNumeric	0	3
21	Median		20.0
22	Mean		57.33
23	Max		152.0
24	Min		0.0
25	Variance		6821.33
26	StdDev		82.59
27	Total		172.0
28			

## 17.12 expression\_match

Finds records in the two tables where the expression evaluates True. This method is usually used only internally, but it is available for general use for those who want it. (in other words, most users generally use other methods like join, col\_match\_same, etc.)

The resulting table shows the matching records indices in table 1 and table 2.

The variable "record1" is set to a record in table1, and "record2" is set to a record in table2. The expression is evaluated for each record in table 1 and table 2.

This function is *\*much\** slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

However, although it is slower, this function can evaluate any expression (including the use of Picalo functions) in the join.

### Signature:

```
<Table> = Simple.expression_match(<Table> table1,
                                   <Table> table2,
                                   <str> expression)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table

- **<str> expression:** The expression to be evaluated, using variables record1 and record2

Returns: A picalo Table containing the table1 match, the table2 match, and the key

### Example:

```

1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ', 2], [1, 'D
3 >>> matches = Simple.expression_match(table1, table2, "record1[0] == record2[0]")
4 >>> matches.view()
5
6 | table1record | table2record |
7 |-----|-----|
8 |           0 |           0 |
9 |           0 |           2 |
10 |-----|-----|

```

## 17.13 find\_duplicates

Finds all duplicates in the given columns. Some audit procedures look for duplicates in columns where duplicates should not exist (such as invoice numbers). This function supports this need.

### Signature:

**<Table> = Simple.find\_duplicates(<Table or TableArray> table)**

- **<Table or TableArray> table:** The table to be analyzed

Returns: A table with one column for the key and one column for the duplicate record indices.

### Example:

```

1 >>> table = Table([( 'col000 ', int), ( 'col001 ', int)], ([1,6000], [2,6000], [2,4000], [3,5000], [
2 >>> dups = Simple.find_duplicates(table, 'col000')
3 >>> dups.view()
4
5 | Key | Rows |
6 |-----|-----|
7 |    2 | [1, 2] |
8 |-----|-----|
9
10 Another example:

```



## 17.14 find\_gaps

Finds gaps in the sequence of values in the given col. A gap is where the value of the column in record1 - the value of the column in record2 does not equal 1.

This function supports audit procedures that look for gaps in ordinarily-sequential column values, such as invoice numbers. Gaps indicate missing values that are normally the target of further research.

Example to find gaps in a set of invoice numbers and amounts >>>  
 table = Table([('col000', int), ('col001', int)], ([1,6000], [2,6000], [3,5000],  
 [5,5000], [6,5000])) >>> gaps = Simple.find\_gaps(table, True, 'col000') >>>  
 gaps.view() +-----+ - Gap Rows - +-----+ - 2 - +-----+ -

### Signature:

```
<Table> = Simple.find_gaps(<Table or TableArray> table,  
                           <bool> ascending,  
                           <str> column)
```

- **<Table or TableArray> table:** The table to be analyzed for gaps in sequence
- **<bool> ascending:** The direction of the sort (True for ascending, False for descending)
- **<str> column:** The column name/index in the table that will be analyzed for gaps in sequence

Returns: A single-columned table of row indices where gaps occur

## 17.15 fuzzycoljoin

Joins two tables based upon a fuzzy match of two column values. The resulting table has rows of both tables that match. This method uses the Simple.join method to join where the fuzzymatch percentage is greater than or equal to the given matchpercent.

### Signature:

```

<Table> = Simple.fuzzycoljoin(<Table> table1,
                             <str> col1,
                             <Table> table2,
                             <str> col2,
                             <float> matchpercent,
                             <int> ngramlen=3,
                             <bool> ignorecase=True)

```

- **<Table> table1:** The first table
- **<str> col1:** The column in the first table to join on
- **<Table> table2:** The second table
- **<str> col2:** The column in the second table to join on
- **<float> matchpercent:** To be joined, col1 and col2 must match by at least this percent
- **<int> ngramlen (optional):** The length of the ngrams to test. Smaller ngrams provide greater tolerance and higher scores. Larger ngrams provide smaller tolerance and lower scores. An ngram of the length of searchtext provides an exact match only. Default is 3. If omitted, defaults to 3.
- **<bool> ignorecase (optional):** Whether to ignore letter case when searching. Default is to ignore case. If omitted, defaults to True.

Returns: The joined table.

### Example:

```

1 >>> table1 = Table([( 'Name', unicode), ( 'Address', unicode)], [['Daniel', '500 West Street'], ['Ma
2 >>> table2 = Table([( 'Name', unicode), ( 'Address', unicode)], [['Steven', '500 West St.'], ['Denny
3 >>> results = Simple.fuzzycolmatch(table1, 'Address', table2, 'Address', 0.30)
4 >>> results.view()
5 +-----+-----+-----+-----+
6 | Name | Address | Name.2 | Add |
7 +-----+-----+-----+-----+
8 | Daniel | 500 West Street | Steven | 500 West St. |
9 +-----+-----+-----+-----+

```

## 17.16 fuzzymatch

Calculates a fuzzy match of text1 and text2. This is an adaptation of the Trigram method found at <http://www.heise.de/ct/english/97/04/386/> by Reinhard Rapp. The score will always be a decimal between 0.0 and 1.0: - 0.0 means the letters in (and sequence of) searchtext is not found at in any form in fulltext. - 1.0 means the letters in searchtext is perfectly found in fulltext.

This algorithm is not as effective as more advanced neural-net-based algorithms, but it is simple and fast. It requires very little processing power compared with more advanced algorithms.

This method is different than fuzzysearch because it expects text1 and text2 to be about the same length. For example, two last names, two addresses, etc.

### Signature:

```
<float> = Simple.fuzzymatch(<str> text1,  
                             <str> text2,  
                             <int> ngramlen=3,  
                             <bool> ignorecase=True)
```

- **<str> text1:** The first text string.
- **<str> text2:** The second text string.
- **<int> ngramlen (optional):** The length of the ngrams to test. Smaller ngrams provide greater tolerance and higher scores. Larger ngrams provide smaller tolerance and lower scores. An ngram of the length of searchtext provides an exact match only. Default is 3. If omitted, defaults to 3.
- **<bool> ignorecase (optional):** Whether to ignore letter case when searching. Default is to ignore case. If omitted, defaults to True.

Returns: A float between 0.0 and 1.0 giving a score for the amount of match.

### Example:

```
1 >>> Simple.fuzzymatch("500 West Street", "500 West St.")  
2 0.69230769230769229
```

## 17.17 fuzzysearch

Calculates a fuzzy search to see if searchtext is found in fulltext. This is an adaptation of the Trigram method found at <http://www.heise.de/ct/english/97/04/386/> by Reinhard Rapp. The score will always be a decimal between 0.0 and 1.0: - 0.0 means the letters in (and sequence of) searchtext is not found at in any form in fulltext. - 1.0 means the letters in searchtext is perfectly found in fulltext.

This algorithm is not as effective as more advanced neural-net-based algorithms, but it is simple and works fairly well.

This method is different than fuzzymatch because it expects a long string for the fulltext variable. For example, searching for a phrase within a large document.

### Signature:

```
<float> = Simple.fuzzysearch(<str> fulltext,  
                             <str> searchtext,  
                             <int> ngramlen=3,  
                             <bool> ignorecase=True)
```

- **<str> fulltext:** The text to find within.
- **<str> searchtext:** The text to search for.
- **<int> ngramlen (optional):** The length of the ngrams to test. Smaller ngrams provide greater tolerance and higher scores. Larger ngrams provide smaller tolerance and lower scores. An ngram of the length of searchtext provides an exact match only. Default is 3. If omitted, defaults to 3.
- **<bool> ignorecase (optional):** Whether to ignore letter case when searching. Default is to ignore case. If omitted, defaults to True.

Returns: A float between 0.0 and 1.0 giving a score for the amount of match.

## 17.18 get\_unordered

Finds all of the records that are out of order as measured by the values of the given cols. The point of this function is not to determine whether a table needs sorting (that would be more inefficient than simply sorting it to begin with). This function supports audit procedures that look for out-of-order items.

### Signature:

```
<Table> = Simple.get_unordered(<Table or TableArray> table,
                               <bool> ascending)
```

- **<Table or TableArray> table:** The table to check for sorting
- **<bool> ascending:** Whether to sort ascending or descending (True for ascending, False for descending)

Returns: A single-column table of the row indices that are out of order

### Example:

```
1 >>> table = Table([( 'col000 ', int), ( 'col001 ', int)], ([5,6], [3,2], [4,4], [4,5], [4,3]))
2 >>> unordered = Simple.get_unordered(table, True, 'col000 ', 'col001 ')
3 >>> unordered.view()
4 +-----+
5 | Unordered Rows |
6 +-----+
7 |                0 |
8 |                3 |
9 +-----+
```

## 17.19 join

Joins two tables together (similar to a regular SQL inner join) where the expression evaluates True.

This function is *\*much\** slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

### Signature:

```
<Table> = Simple.join(<Table> table1,
                      <Table> table2,
                      <str> expression)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table
- **<str> expression:** The expression to be evaluated, using variables record1 and record2

Returns: A picalo Table containing matching records from the two source tables

### Example:

```
1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ',2], [1, 'D
3 >>> matches = Simple.join(table1, table2, "record1[0] == record2[0]")
4 >>> matches.view()
5 +-----+-----+-----+-----+-----+
6 | col000 | col001 | col000_2 | col001_2 | col002 |
7 +-----+-----+-----+-----+-----+
8 |      3 |      2 |      3 | Bailey   |      2 |
9 |      3 |      2 |      3 | Sally    |      2 |
10 +-----+-----+-----+-----+-----+
```

## 17.20 left\_join

Joins two tables together (similar to a regular SQL LEFT join) where the expression evaluates True. All records in the first table are returned, and only matching records in the second table are returned.

This function is \*much\* slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

### Signature:

```
<Table> = Simple.left_join(<Table> table1,
                           <Table> table2,
                           <str> expression)
```

- **<Table> table1:** The first source table

- **<Table> table2:** The second source table
- **<str> expression:** The expression to be evaluated, using variables record1 and record2

Returns: A picalo Table containing matching records from the two source tables

### Example:

```

1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ', 2], [1, 'Da
3 >>> matches = Simple.left_join(table1, table2, "record1[0] == record2[0]")
4 >>> matches.view()
5 +-----+
6 | col000 | col001 | col000_1 | col001_1 | col002 |
7 +-----+
8 |      3 |      2 |      3   | Bailey   |      2   |
9 |      3 |      2 |      3   | Sally    |      2   |
10 |      4 |      5 |      <N>  | <N>      | <N>      |
11 +-----+

```

## 17.21 regex\_match

Returns whether the given value matches the given regular expression pattern. Regular expressions are a very powerful matching language that are available in many computer languages and programs. This function uses the Python re module internally, and it follows the re rules. See <http://docs.python.org/dev/howto/regex> for a tutorial on regular expressions.

The function only returns True or False. To discover the actual matched text or groups within text, use the Python re module directly.

### Signature:

```

<bool> = Simple.regex_match(<str> pattern,
                             <str> value,
                             <bool> ignorecase=False)

```

- **<str> pattern:** A regular expression pattern.
- **<str> value:** The value to match against the pattern. If not a string, it is converted automatically so matching is possible.

- `<bool> ignorecase` (optional): Whether to make the match case sensitive (the default) or not. If omitted, defaults to `False`.

Returns: `True` if the value matches the pattern, `False` otherwise.

### Example 1:

```
1 >>> Simple.regex_match('^.+@.+\\.\\w{2,4}$', 'someone@myemail.info')
2 True
```

### Example 2:

```
1 >>> Simple.regex_match('1-\\d{3}-\\d{3}-\\d{4}', '1-800-555-1234')
2 True
```

### Example 3:

```
1 >>> pat = '^P(ost){0,1}\\.{0,1} *O(ffice){0,1}\\.{0,1} *Box$'
2 >>> Simple.regex_match(pat, 'PO box', True)
3 True
4 >>> Simple.regex_match(pat, 'P.O. Box', True)
5 True
6 >>> Simple.regex_match(pat, 'Post Office Box', True)
7 True
8 >>> Simple.regex_match(pat, 'po box road', True)
9 False
```

## 17.22 right\_join

Joins two tables together (similar to a regular SQL RIGHT join) where the expression evaluates `True`. All records in the second table are returned, and only matching records in the first table are returned.

This function is *\*much\** slower than a Database join. If you are getting data from a database, use the SQL join instead. This method is provided when you need to join tables that were loaded from CSV, etc.

### Signature:

```
<Table> = Simple.right_join(<Table> table1,
                           <Table> table2,
                           <str> expression)
```

- **<Table> table1:** The first source table
- **<Table> table2:** The second source table



- **<str> expression:** The expression to be evaluated, using variables record1 and record2

Returns: A picalo Table containing matching records from the two source tables

### Example:

```

1 >>> table1 = Table([( 'col000 ', int), ( 'col001 ', int)], ([3,2], [4,5]))
2 >>> table2 = Table([( 'col000 ', int), ( 'col001 ', unicode), ( 'col002 ', int)], ([3, 'Bailey ', 2], [1, 'Da
3 >>> matches = Simple.right_join(table1, table2, "record1[0] == record2[0]")
4 >>> matches.view()
5 +-----+-----+-----+-----+-----+
6 | col000 | col001 | col002 | col000_1 | col001_1 |
7 +-----+-----+-----+-----+-----+
8 |      3 | Bailey |      2 |      3 |      2 |
9 |      1 | Dan   |      2 | <N>    | <N>    |
10 |      3 | Sally |      2 |      3 |      2 |
11 +-----+-----+-----+-----+-----+

```

## 17.23 select

Selects records from a table based upon a custom expression and returns a new table including only those records.

The expression should evaluate using "record" for the current record and the column names for individual column values. The expression should evaluate to True or False, as in (assuming recid is a column name): "recid < 1000"

The select function is very slow compared with database SELECT statements. If you have the choice (e.g. if you are using a database data source), use the SQL SELECT instead of this function. This is useful when data comes from sources other than SQL, such as CSV/TSV files.

### Signature:

```

<Table> = Simple.select(<Table or TableArray> table,
                        <str> expression)

```

- **<Table or TableArray> table:** The table records will be selected from
- **<str> expression:** An expression that evaluates each record and returns whether each should be included in the results table.

Returns: A new table containing the records for which func evaluated to true (1)

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([5,6], [3,2], [4,6]))
2 >>> results = Simple.select(table, "col001 > 5")
3 >>> results.view()
4 +-----+-----+
5 | col000 | col001 |
6 +-----+-----+
7 |      5 |      6 |
8 |      4 |      6 |
9 +-----+-----+
```

## 17.24 select\_by\_value

Selects record from a table given colname=value pairs and returns a new table including only those records.

This method *is* efficient and can be used often. It calculates indices as needed and should select very fast.

If you need to do more complex selection, such as where a field is greater than some value, use Simple.select, which allows the use of arbitrary (and possibly powerful) expressions.

### Signature:

```
<Table> = Simple.select_by_value(<Table or TableArray> table)
```

- **<Table or TableArray> table:** The table records will be selected from

Returns: A new table containing the records with matching key(s).

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int), ('col002', unicode)], [
2 ...     [5,6,'flo'],
3 ...     [3,2,'sally'],
4 ...     [4,6,'dan'],
5 ...     [4,7,'stu'],
6 ...     [4,7,'ben']
7 ... ])
8 >>> results = Simple.select_by_value(table, col001=6, col002='dan')
9 >>> results.view()
```

10							
11		col000		col001		col002	
12							
13		4		6		dan	
14							

## 17.25 select\_nonoutliers

A convenience function to select non-outliers from a table. Outliers are those with column values below the min or above the max. The source table is not modified.

This type of filtering could also be done with the Simple.select method, but since filtering of outliers is so common, it is given its own function.

### Signature:

```
<Table> = Simple.select_nonoutliers(<Table or TableArray> table,
                                     <str> col,
                                     <object> min=None,
                                     <object> max=None)
```

- **<Table or TableArray> table:** The table to be filtered
- **<str> col:** The column index to evaluate
- **<object> min (optional):** An optional value that specifies a lower bound for included records. If omitted, no lower bound is used. If omitted, defaults to None.
- **<object> max (optional):** An optional value that specifies an upper bound for included records. If omitted, no upper bound is used. If omitted, defaults to None.

Returns: A new table with all outliers removed

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([8,6], [3,2], [0,4], [4,6]))
2 >>> selected = Simple.select_nonoutliers(table, 'col000', min=2, max=5)
3 >>> selected.view()
4 |-----|
5 | col000 | col001 |
6 |-----|
```

7		3		2	
8		4		6	
9	+	-----	+	-----	+

## 17.26 select\_nonoutliers\_z

A convenience function to select non-outliers from a table. Outliers are those with column values with zscores above +zscore or below -zscore. The source table is not modified.

This type of filtering could also be done with the Simple.select method, but since filtering of outliers is so common, it is given its own function.

### Signature:

```
<Table> = Simple.select_nonoutliers_z(<Table> table,
                                     <str> col,
                                     <float> zscore)
```

- **<Table> table:** The table to be filtered
- **<str> col:** The column index to evaluate
- **<float> zscore:** The zscore to filter above and below

Returns: A new table containing only the outliers

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([8,8], [3,2], [0,4], [4,3]))
2 >>> selected = Simple.select_nonoutliers_z(table, 'col001', 1)
3 >>> selected.view()
4 +-----+-----+
5 | col000 | col001 |
6 +-----+-----+
7 |      3 |      2 |
8 |      0 |      4 |
9 |      4 |      3 |
10 +-----+-----+
```

## 17.27 select\_outliers

A convenience function to select outliers from a table. Outliers are those with column values below the min or above the max. The source table is not modified.

This type of filtering could also be done with the Simple.select method, but since filtering of outliers is so common, it is given its own function.

### Signature:

```
<Table> = Simple.select_outliers(<Table or TableArray> table,
                                <str> col,
                                <object> min=None,
                                <object> max=None)
```

- **<Table or TableArray> table:** The table to be filtered
- **<str> col:** The column index to evaluate
- **<object> min (optional):** An optional value that specifies a lower bound for included records. If omitted, no lower bound is used. If omitted, defaults to None.
- **<object> max (optional):** An optional value that specifies an upper bound for included records. If omitted, no upper bound is used. If omitted, defaults to None.

Returns: A new table containing only the outliers

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([8,6], [3,2], [0,4], [4,6]))
2 >>> selected = Simple.select_outliers(table, 'col000', min=2, max=5)
3 >>> selected.view()
4 +-----+-----+
5 | col000 | col001 |
6 +-----+-----+
7 |      8 |      6 |
8 |      0 |      4 |
9 +-----+-----+
```

## 17.28 select\_outliers\_z

A convenience function to select outliers from a table. Outliers are those with column values with zscores above +zscore or below -zscore. The source table is not modified.

This type of filtering could also be done with the Simple.select method, but since filtering of outliers is so common, it is given its own function.

### Signature:

```
<Table> = Simple.select_outliers_z(<Table or TableArray> table,
                                   <str> col,
                                   <float> zscore)
```

- **<Table or TableArray> table:** A Picalo table
- **<str> col:** The column index to evaluate
- **<float> zscore:** The zscore to filter above and below

Returns: A new table containing only the outliers

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([8,8], [3,2], [0,4], [4,3]))
2 >>> selected = Simple.select_outliers_z(table, 'col001', 1)
3 >>> selected.view()
4 +-----+-----+
5 | col000 | col001 |
6 +-----+-----+
7 |      8 |      8 |
8 +-----+-----+
```

## 17.29 select\_records

Selects records from a table given a number of table record indices and returns a new table including only those records.

This could obviously be done with a few lines of code, but this function provides the functionality for more readable code.

This method *is* efficient and can be used often.

### Signature:

```
<Table> = Simple.select_records(<Table or TableArray> table,
                                <list> record_indices)
```

- **<Table or TableArray> table:** The table records will be selected from
- **<list> record\_indices:** A list of indices (of type int) of the records to be included.

Returns: A new table containing the records included in the record\_indices list.

### Example:

```
1 >>> table = Table([('col000', int), ('col001', int)], ([5,6], [3,2], [4,6]))
2 >>> results = Simple.select_records(table, [0,2])
3 >>> results.view()
4 +-----+
5 | col000 | col001 |
6 +-----+
7 |      5 |      6 |
8 |      4 |      6 |
9 +-----+
```

## 17.30 sort

Sorts a table by values in one or more columns.

### Signature:

```
Simple.sort(<Table or TableArray> table,
            <bool> ascending)
```

- **<Table or TableArray> table:** The table to be sorted.
- **<bool> ascending:** Sorts ascending when True, descending when False

### Example:

```

1 >>> table = Table([('col000', int), ('col001', int)], ([5,6], [3,2], [4,6]))
2 >>> Simple.sort(table, True, 'col000', 'col001')
3 >>> table.view()
4 +-----+
5 | col000 | col001 |
6 +-----+
7 |      3 |      2 |
8 |      4 |      6 |
9 |      5 |      6 |
10 +-----+

```

## 17.31 soundex

Calculates a soundex computation for the given text. Soundex is a standard algorithm for comparing text in a fuzzy way. For example, it sees 'Smith', 'Smoth', and 'Smiith' as the same thing. From a fraud detection perspective, it is extremely useful to match addresses, employee names, vendor names, and other text that may have variations in it.

Soundex creates a number out of text. To compare two text values, compute a soundex hash of both values and compare the soundex results.

Note that soundex is optimized for English names. If you need to optimize for other languages, search the Internet for an appropriate set of digits.

Also note that this method calculates the raw soundex score. It may be more useful to use `Simple.soundexcol`, which runs the soundex algorithm on an entire column.

Credits: Taken from ASPN: implementation 2000-12-24 by Gregory Jorgensen License: Public domain by G.J.

### Signature:

```

<str> = Simple.soundex(<str> text,
                      <int> len=4,
                      <str> digits="01230120022455012623010202")

```

- **<str> text:** The text to compute soundex on
- **<int> len (optional):** The length of the resulting soundex hash. Longer lengths are more precise. Shorter lengths are more fuzzy. If omitted, defaults to 4.



- **<str> digits (optional):** The digits to use in the soundex algorithm. The default digits are optimized for English names. If omitted, defaults to 01230120022455012623010202.

Returns: The soundex hash for the given text.

### Example:

```
1 >>> Simple.soundex('Smith')
2 'S530'
3 >>> Simple.soundex('Smoth')
4 'S530'
5 >>> Simple.soundex('Smithinson', len=6)
6 'S53525'
7 >>> Simple.soundex('Smithinall', len=6)
8 'S53540'
```

## 17.32 soundexcol

Calculates the soundex score for each value in a column and appends the scores as a new column in the table. The new column is named 'column\_soundex' (where column is the name of the column).

See the Simple.soundex method for information on the soundex algorithm.

### Signature:

```
Simple.soundexcol(<Table> table,
                  <str> col)
```

- **<Table> table:** A Picalo table
- **<str> col:** The column name/index to calculate the soundex score on.

### Example:

```
1 >>> t = Table([('name', unicode)], [['Samuel'], ['Sally'], ['Max'], ['Maxx']])
2 >>> Simple.soundexcol(t, 'name')
3 >>> t.view()
4 +-----+
5 |  name  | hash |
6 +-----+
7 | Samuel | S540 |
8 | Sally  | S400 |
9 | Max    | M200 |
10 | Maxx   | M200 |
11 +-----+
12 >>> # Grouping by the new 'hash' column will quickly place duplicates together in groups.
```

## 17.33 transpose

Transposes the table, which means the columns and rows are switched. A transposed (inverted) table is returned.

Be careful with transposing large tables – the new table will have as many columns as the source table has rows. Large tables are often unmanageable because of the large number of columns they produce.

Since Picalo column names must conform to a specific format and must be unique, some changes may be made to the values in the transposition process.

Since Picalo cannot guess the new column types, all columns are typed as unicode columns. Use `table.set_type` to set the types after transposition.

### Signature:

```
<Table> = Simple.transpose(<Table> table)
```

- **<Table> table:** The table to be transposed

Returns: A new table containing the transposed values

## 17.34 wildcard\_match

Returns whether the given value matches the given wildcard pattern. This is the simplest way to match values to patterns in Picalo. It supports only three special characters:

- A question mark (?) matches a single letter (A-Z and a-z).
- A pound sign (#) matches a single number (0-9).
- A star (\*) matches zero or more letters or numbers.

Internally, the function turns the wildcard characters into the appropriate regular expression and uses the Python `re` module to perform the match.

This function only returns True or False. To discover the actual matched text or groups within text, use the Python `re` module directly.

For a more advanced and powerful method of pattern matching, see the `Simple.regex_match` function, which allows matching via the full regular expression language.

### Signature:

```
<bool> = Simple.wildcard_match(<str> pattern,  
                                <str> value,  
                                <bool> ignorecase=False,  
                                <bool> fullmatch=True)
```

- **<str> pattern:** A regular expression pattern.
- **<str> value:** The value to match against the pattern. If not a string, it is converted automatically so matching is possible.
- **<bool> ignorecase (optional):** Whether to make the match case sensitive (the default) or not. If omitted, defaults to False.
- **<bool> fullmatch (optional):** Whether to force matchin of the entire value (the default) or to allow partial matching within the value string. If omitted, defaults to True.

Returns: True if the value matches the pattern, False otherwise.

### Example 1:

```
1 >>> Simple.wildcard_match('?*@?*.??*', 'someone@myemail.info')  
2 True
```

### Example 2:

```
1 >>> Simple.wildcard_match('1-###-###-####', '1-800-555-1234')  
2 True
```

### Example 3:

```
1 >>> Simple.wildcard_match('#####-##-##-##:###:###:###', '2009-12-25 15:33:01.155')  
2 True
```

# Chapter 18

## string

A collection of string operations (most are no longer used).

Warning: most of the code you see here isn't normally used nowadays. Beginning with Python 1.6, many of these functions are implemented as methods on the standard string object. They used to be implemented by a built-in module called `strop`, but `strop` is now obsolete itself.

Public module variables:

`whitespace` – a string containing all characters considered whitespace  
`lowercase` – a string containing all characters considered lowercase letters  
`uppercase` – a string containing all characters considered uppercase letters  
`letters` – a string containing all characters considered letters  
`digits` – a string containing all characters considered decimal digits  
`hexdigits` – a string containing all characters considered hexadecimal digits  
`octdigits` – a string containing all characters considered octal digits  
`punctuation` – a string containing all characters considered punctuation  
`printable` – a string containing all characters considered printable

### 18.1 `atof`

`atof(s)` -> float

Return the floating point number represented by the string `s`.

**Signature:**

`string.atof(<object> s)`

- `<object> s`:

## 18.2 atoi

atoi(s [,base]) -> int

Return the integer represented by the string s in the given base, which defaults to 10. The string s must consist of one or more digits, possibly preceded by a sign. If base is 0, it is chosen from the leading characters of s, 0 for octal, 0x or 0X for hexadecimal. If base is 16, a preceding 0x or 0X is accepted.

### Signature:

```
string.atoi(<object> s,  
            <object> base=10)
```

- <object> s:
- <object> base (optional): If omitted, defaults to 10.

## 18.3 atol

atol(s [,base]) -> long

Return the long integer represented by the string s in the given base, which defaults to 10. The string s must consist of one or more digits, possibly preceded by a sign. If base is 0, it is chosen from the leading characters of s, 0 for octal, 0x or 0X for hexadecimal. If base is 16, a preceding 0x or 0X is accepted. A trailing L or l is not accepted, unless base is 0.

### Signature:

```
string.atol(<object> s,  
            <object> base=10)
```

- <object> s:
- <object> base (optional): If omitted, defaults to 10.

## 18.4 capitalize

`capitalize(s) -> string`

Return a copy of the string `s` with only its first character capitalized.

**Signature:**

`string.capitalize(<object> s)`

- `<object> s`:

## 18.5 capwords

`capwords(s, [sep]) -> string`

Split the argument into words using `split`, capitalize each word using `capitalize`, and join the capitalized words using `join`. Note that this replaces runs of whitespace characters by a single space.

**Signature:**

`string.capwords(<object> s,  
                  <object> sep=None)`

- `<object> s`:
- `<object> sep` (optional): If omitted, defaults to `None`.

## 18.6 center

`center(s, width[, fillchar]) -> string`

Return a center version of `s`, in a field of the specified width. padded with spaces as needed. The string is never truncated. If specified the `fillchar` is used instead of spaces.

**Signature:**

`string.center(<object> s,  
              <object> width)`

- `<object> s`:
- `<object> width`:

## 18.7 count

`count(s, sub[, start[,end]]) -> int`

Return the number of occurrences of substring `sub` in string `s[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

**Signature:**

`string.count(<object> s)`

- `<object> s`:

## 18.8 expandtabs

`expandtabs(s [,tabsize]) -> string`

Return a copy of the string `s` with all tab characters replaced by the appropriate number of spaces, depending on the current column, and the `tabsize` (default 8).

**Signature:**

`string.expandtabs(<object> s,  
                  <object> tabsize=8)`

- `<object> s`:
- `<object> tabsize` (optional): If omitted, defaults to 8.

## 18.9 find

`find(s, sub [,start [,end]]) -> in`

Return the lowest index in `s` where substring `sub` is found, such that `sub` is contained within `s[start,end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Return -1 on failure.

**Signature:**

`string.find(<object> s)`

- `<object> s`:

## 18.10 `Formatter.check_unused_args`

None

**Signature:**

```
[Formatter].check_unused_args(<object> used_args,  
                             <object> args,  
                             <object> kwargs)
```

- `<object> used_args`:
- `<object> args`:
- `<object> kwargs`:

## 18.11 `Formatter.convert_field`

None

**Signature:**

```
[Formatter].convert_field(<object> value,  
                          <object> conversion)
```

- `<object> value`:
- `<object> conversion`:

## 18.12 `Formatter.format`

None

**Signature:**

```
[Formatter].format(<object> format_string)
```

- `<object> format_string`:



## 18.13 `Formatter.format_field`

None

**Signature:**

```
[Formatter].format_field(<object> value,  
                        <object> format_spec)
```

- `<object> value`:
- `<object> format_spec`:

## 18.14 `Formatter.get_field`

None

**Signature:**

```
[Formatter].get_field(<object> field_name,  
                    <object> args,  
                    <object> kwargs)
```

- `<object> field_name`:
- `<object> args`:
- `<object> kwargs`:

## 18.15 `Formatter.get_value`

None

**Signature:**

```
[Formatter].get_value(<object> key,  
                    <object> args,  
                    <object> kwargs)
```

- `<object> key`:
- `<object> args`:
- `<object> kwargs`:

## 18.16 `Formatter.parse`

None

**Signature:**

```
[Formatter].parse(<object> format_string)
```

- `<object> format_string:`

## 18.17 `Formatter.vformat`

None

**Signature:**

```
[Formatter].vformat(<object> format_string,  
                   <object> args,  
                   <object> kwargs)
```

- `<object> format_string:`
- `<object> args:`
- `<object> kwargs:`

## 18.18 `index`

`index(s, sub [,start [,end]]) -> int`

Like `find` but raises `ValueError` when the substring is not found.

**Signature:**

```
string.index(<object> s)
```

- `<object> s:`

## 18.19 join

`join(list [,sep]) -> string`

Return a string composed of the words in list, with intervening occurrences of sep. The default separator is a single space.

(joinfields and join are synonymous)

### Signature:

```
string.join(<object> words,  
           <object> sep= )
```

- **<object> words:**
- **<object> sep (optional):** If omitted, defaults to .

## 18.20 joinfields

`join(list [,sep]) -> string`

Return a string composed of the words in list, with intervening occurrences of sep. The default separator is a single space.

(joinfields and join are synonymous)

### Signature:

```
string.joinfields(<object> words,  
                 <object> sep= )
```

- **<object> words:**
- **<object> sep (optional):** If omitted, defaults to .

## 18.21 ljust

`ljust(s, width[, fillchar]) -> string`

Return a left-justified version of s, in a field of the specified width, padded with spaces as needed. The string is never truncated. If specified the fillchar is used instead of spaces.

### Signature:

```
string.ljust(<object> s,  
            <object> width)
```

- <object> s:
- <object> width:

## 18.22 lower

lower(s) -> string

Return a copy of the string s converted to lowercase.

**Signature:**

```
string.lower(<object> s)
```

- <object> s:

## 18.23 lstrip

lstrip(s [,chars]) -> string

Return a copy of the string s with leading whitespace removed. If chars is given and not None, remove characters in chars instead.

**Signature:**

```
string.lstrip(<object> s,  
             <object> chars=None)
```

- <object> s:
- <object> chars (optional): If omitted, defaults to None.

## 18.24 replace

`replace(str, old, new[, maxsplit]) -> string`

Return a copy of string `str` with all occurrences of substring `old` replaced by `new`. If the optional argument `maxsplit` is given, only the first `maxsplit` occurrences are replaced.

### Signature:

```
string.replace(<object> s,  
               <object> old,  
               <object> new,  
               <object> maxsplit=-1)
```

- **<object> s:**
- **<object> old:**
- **<object> new:**
- **<object> maxsplit (optional):** If omitted, defaults to -1.

## 18.25 rfind

`rfind(s, sub [,start [,end]]) -> int`

Return the highest index in `s` where substring `sub` is found, such that `sub` is contained within `s[start,end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Return -1 on failure.

### Signature:

```
string.rfind(<object> s)
```

- **<object> s:**

## 18.26 rindex

`rindex(s, sub [,start [,end]]) -> int`

Like `rfind` but raises `ValueError` when the substring is not found.

**Signature:**

```
string.rindex(<object> s)
```

- `<object> s`:

## 18.27 rjust

`rjust(s, width[, fillchar]) -> string`

Return a right-justified version of `s`, in a field of the specified width, padded with spaces as needed. The string is never truncated. If specified the `fillchar` is used instead of spaces.

**Signature:**

```
string.rjust(<object> s,  
            <object> width)
```

- `<object> s`:
- `<object> width`:

## 18.28 rsplit

`rsplit(s [,sep [,maxsplit]]) -> list of strings`

Return a list of the words in the string `s`, using `sep` as the delimiter string, starting at the end of the string and working to the front. If `maxsplit` is given, at most `maxsplit` splits are done. If `sep` is not specified or is `None`, any whitespace string is a separator.

**Signature:**

```
string.rsplit(<object> s,  
             <object> sep=None,  
             <object> maxsplit=-1)
```

- **<object> s:**
- **<object> sep** (optional): If omitted, defaults to None.
- **<object> maxsplit** (optional): If omitted, defaults to -1.

## 18.29 rstrip

`rstrip(s [,chars]) -> string`

Return a copy of the string `s` with trailing whitespace removed. If `chars` is given and not None, remove characters in `chars` instead.

### Signature:

```
string.rstrip(<object> s,  
              <object> chars=None)
```

- **<object> s:**
- **<object> chars** (optional): If omitted, defaults to None.

## 18.30 split

`split(s [,sep [,maxsplit]]) -> list of strings`

Return a list of the words in the string `s`, using `sep` as the delimiter string. If `maxsplit` is given, splits at no more than `maxsplit` places (resulting in at most `maxsplit+1` words). If `sep` is not specified or is None, any whitespace string is a separator.

(`split` and `splitfields` are synonymous)

### Signature:

```
string.split(<object> s,  
            <object> sep=None,  
            <object> maxsplit=-1)
```

- **<object> s:**
- **<object> sep** (optional): If omitted, defaults to None.
- **<object> maxsplit** (optional): If omitted, defaults to -1.

## 18.31 splitfields

`split(s [,sep [,maxsplit]])` -> list of strings

Return a list of the words in the string `s`, using `sep` as the delimiter string. If `maxsplit` is given, splits at no more than `maxsplit` places (resulting in at most `maxsplit+1` words). If `sep` is not specified or is `None`, any whitespace string is a separator.

(`split` and `splitfields` are synonymous)

### Signature:

```
string.splitfields(<object> s,  
                  <object> sep=None,  
                  <object> maxsplit=-1)
```

- **<object> s:**
- **<object> sep** (optional): If omitted, defaults to `None`.
- **<object> maxsplit** (optional): If omitted, defaults to `-1`.

## 18.32 strip

`strip(s [,chars])` -> string

Return a copy of the string `s` with leading and trailing whitespace removed. If `chars` is given and not `None`, remove characters in `chars` instead. If `chars` is unicode, `S` will be converted to unicode before stripping.

### Signature:

```
string.strip(<object> s,  
            <object> chars=None)
```

- **<object> s:**
- **<object> chars** (optional): If omitted, defaults to `None`.



### 18.33 swapcase

swapcase(s) -> string

Return a copy of the string s with upper case characters converted to lowercase and vice versa.

**Signature:**

string.swapcase(<object> s)

- <object> s:

### 18.34 Template

None

**Signature:**

string.Template(<object> template)

- <object> template:

### 18.35 Template.safe\_substitute

None

**Signature:**

[Template].safe\_substitute()

### 18.36 Template.substitute

None

**Signature:**

[Template].substitute()

## 18.37 translate

`translate(s,table [,deletions]) -> string`

Return a copy of the string `s`, where all characters occurring in the optional argument `deletions` are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256. The `deletions` argument is not allowed for Unicode strings.

### Signature:

```
string.translate(<object> s,  
                <object> table,  
                <object> deletions=)
```

- **<object> s:**
- **<object> table:**
- **<object> deletions (optional):** If omitted, defaults to `.`

## 18.38 upper

`upper(s) -> string`

Return a copy of the string `s` converted to uppercase.

### Signature:

```
string.upper(<object> s)
```

- **<object> s:**

## 18.39 zfill

`zfill(x, width) -> string`

Pad a numeric string `x` with zeros on the left, to fill a field of the specified width. The string `x` is never truncated.

### Signature:

```
string.zfill(<object> x,  
            <object> width)
```

- **<object> x:**
- **<object> width:**

# Chapter 19

## sys

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

`argv` – command line arguments; `argv[0]` is the script pathname if known  
`path` – module search path; `path[0]` is the script directory, else ”  
`modules` – dictionary of loaded modules

`displayhook` – called to show results in an interactive session  
`excepthook` – called to handle any uncaught exception other than `SystemExit`  
To customize printing in an interactive session or to install a custom top-level exception handler, assign other functions to replace these.

`exitfunc` – if `sys.exitfunc` exists, this routine is called when Python exits  
Assigning to `sys.exitfunc` is deprecated; use the `atexit` module instead.

`stdin` – standard input file object; used by `raw_input()` and `input()`  
`stdout` – standard output file object; used by the `print` statement  
`stderr` – standard error object; used for error messages  
By assigning other file objects (or objects that behave like files) to these, it is possible to redirect all of the interpreter’s I/O.

`last_type` – type of last uncaught exception  
`last_value` – value of last uncaught exception  
`last_traceback` – traceback of last uncaught exception  
These three are only available in an interactive session after a traceback has been printed.

`exc_type` – type of exception currently being handled  
`exc_value` – value of exception currently being handled  
`exc_traceback` – traceback of exception currently being handled  
The function `exc_info()` should be used instead of these three, because it is thread-safe.

Static objects:

`maxint` – the largest supported integer (the smallest is `-maxint-1`) `maxsize` – the largest supported length of containers. `maxunicode` – the largest supported character `builtin_module_names` – tuple of module names built into this interpreter `version` – the version of this interpreter as a string `version_info` – version information as a tuple `hexversion` – version information encoded as a single integer `copyright` – copyright notice pertaining to this interpreter `platform` – platform identifier `executable` – pathname of this Python interpreter `prefix` – prefix used to find the Python library `exec_prefix` – prefix used to find the machine-specific Python library `__stdin__` – the original stdin; don't touch! `__stdout__` – the original stdout; don't touch! `__stderr__` – the original stderr; don't touch! `__displayhook__` – the original displayhook; don't touch! `__excepthook__` – the original excepthook; don't touch!

Functions:

`displayhook()` – print an object to the screen, and save it in `__builtin__` `excepthook()` – print an exception and its traceback to `sys.stderr` `exc_info()` – return thread-safe information about the current exception `exc_clear()` – clear the exception state for the current thread `exit()` – exit the interpreter by raising `SystemExit` `getdlopenflags()` – returns flags to be used for `dlopen()` calls `getprofile()` – get the global profiling function `getrefcount()` – return the reference count for an object (plus one :-)) `getrecursionlimit()` – return the max recursion depth for the interpreter `getsizeof()` – return the size of an object in bytes `gettrace()` – get the global debug tracing function `setcheckinterval()` – control how often the interpreter checks for events `setdlopenflags()` – set the flags to be used for `dlopen()` calls `setprofile()` – set the global profiling function `setrecursionlimit()` – set the max recursion depth for the interpreter `settrace()` – set the global debug tracing function

# Chapter 20

## Trending

The Trending module contains functions that highlight trends in data. Since fraud is most often found in changes over time, this module is useful in looking at trends over time.

### 20.1 average\_slope

Computes the average of the slopes between the points given. If xcol is None, it is generated starting as 0, 1, 2, 3, etc.

**Signature:**

```
<Table> = Trending.average_slope(<Table> table,  
                                <str> ycol,  
                                <str> xcol="None")
```

- **<Table> table:** The table to calculate the slope on
- **<str> ycol:** The y column to use. The maximum and minimum are taken from this column.
- **<str> xcol (optional):** The x column to use. This is optional. If omitted, defaults to None.

Returns: The average slope between points

**Example:**

```

1 >>> from picalo import *
2 >>> table = Table([('col000', unicode), ('col001', int), ('col002', int)], [
3     ['Dan', 10, 8],
4     ['Sally', 12, 12],
5     ['Dan', 11, 15],
6     ['Sally', 12, 14],
7     ['Dan', 11, 16],
8     ['Sally', 15, 15],
9     ['Dan', 16, 15],
10    ['Sally', 13, 14]])
11 >>> results = Trending.average_slope(table, 2, 1)
12 >>> results.view()
13 +-----+
14 | Average Slope |
15 +-----+
16 | -0.559523809524 |
17 +-----+

```

## 20.2 cusum

Calculates a cusum, a cumulative difference in the values of a list at each row in the table. The cusum calculation gives a sense of the overall direction of a curve.

### Signature:

```
<Table> = Trending.cusum(<Table or TableArray> table,
                        <str> col)
```

- **<Table or TableArray> table:** The table to be cusumed.
- **<str> col:** The column name or index to cusum.

Returns: A new table containing a single column for the cusum value.

### Example:

```

1 >>> table = Table([('col000', int), ('col001', int)], ([5,6], [3,2], [4,6]))
2 >>> cusum = Trending.cusum(table, 0) # cusum the first column (5, 3, 4)
3 >>> cusum.view()
4 +-----+
5 | col000_cusum |
6 +-----+
7 |           0 |
8 |          -2 |
9 |          -1 |
10 +-----+

```

## 20.3 handshake\_slope

Computes the slope between every point given. If xcol is None, it is generated starting as 0, 1, 2, 3, etc.

For example: Assume 5 points. The slopes from points 1 to 2, 1 to 3, 1 to 4, 1 to 5, 2 to 3, 2 to 4, 2 to 5, 3 to 4, 3 to 5, and 4 to 5 are calculated. The sum of those slopes are divided by the total number of points to get an idea of the general trend.

### Signature:

```
<Table> = Trending.handshake_slope(<Table> table,
                                   <str> ycol,
                                   <str> xcol="None")
```

- **<Table> table:** The table to calculate the handshake slope on.
- **<str> ycol:** The y column to use.
- **<str> xcol (optional):** The x column to use. This is optional. If omitted, defaults to None.

Returns: A picalo table containing one cell: the calculated slope.

## 20.4 highlow\_slope

Computes a slope based on the minimum Y and the X that goes with it and the maximum Y and the X that goes with it. Returns the X that goes with the minimum Y, the minimum Y, the X that goes with the maximum Y, the maximum Y, and the slope.

If xcol is None, it is generated starting as 0, 1, 2, 3, etc.

### Signature:

```
<Table> = Trending.highlow_slope(<Table> table,
                                 <str> ycol,
                                 <str> xcol="None")
```

- **<Table> table:** The table to calculate the slope on



- **<str> ycol:** The y column to use. The maximum and minimum are taken from this column.
- **<str> xcol (optional):** The x column to use. This is optional. If omitted, defaults to None.

Returns: A table with the first record giving the x and y of the minimum y, the x and y of the maximum y, and the slope between the two points.

### Example:

```

1 >>> table = Table([( 'col000 ', unicode), ( 'col001 ', int), ( 'col002 ', int)], [
2         [ 'Dan ', 10, 8],
3         [ 'Sally ', 12, 12],
4         [ 'Dan ', 11, 15],
5         [ 'Sally ', 12, 14],
6         [ 'Dan ', 11, 16],
7         [ 'Sally ', 15, 15],
8         [ 'Dan ', 16, 15],
9         [ 'Sally ', 13, 14]])
10 >>> results = Trending.highlow_slope(table, 2, 1)
11 >>> results.view()
12 +-----+
13 | MinX | MinY | MaxX | MaxY | Slope |
14 +-----+
15 |   10 |    8 |   11 |   16 |   8.0 |
16 +-----+

```

## 20.5 regression

Computes the regressionline for the points given. Returns the slope, intercept, correlation, and r-squared value of the regression line for the Points If xcol is None, it is generated starting as 0, 1, 2, 3, etc.

### Signature:

```

<Table> = Trending.regression(<Table> table,
                             <str> ycol,
                             <str> xcol="None")

```

- **<Table> table:** The table to calculate the simple regression on
- **<str> ycol:** The y column to use
- **<str> xcol (optional):** The x column to use. This is optional. If omitted, defaults to None.

Returns: A table containing the slope, intercept, correlation, and rSquared

### Example:

```

1 >>> from picalo import *
2 >>> table = Table([('col000', unicode), ('col001', int), ('col002', int)], [
3     ['Dan', 10, 8],
4     ['Sally', 12, 12],
5     ['Dan', 11, 15],
6     ['Sally', 12, 14],
7     ['Dan', 11, 16],
8     ['Sally', 15, 15],
9     ['Dan', 16, 15],
10    ['Sally', 13, 14]])
11 >>> results = Trending.regression(table, 1)
12 >>> results.view()
13 +-----+-----+-----+-----+
14 |      Slope      | Intercept | Correlation |  RSquared  |
15 +-----+-----+-----+-----+
16 | 0.619047619048 | 10.3333333333 | 0.0428888771398 | 0.00183945578231 |
17 +-----+-----+-----+-----+

```

# Chapter 21

## urllib

Open an arbitrary URL.

See the following document for more info on URLs: "Names and Addresses, URIs, URLs, URNs, URCs", at <http://www.w3.org/pub/WWW/Addressing/Overview.htm>

See also the HTTP spec (from which the error codes are derived): "HTTP - Hypertext Transfer Protocol", at <http://www.w3.org/pub/WWW/Protocols/>

Related standards and specs: - RFC1808: the "relative URL" spec. (authoritative status) - RFC1738 - the "URL standard". (authoritative status) - RFC1630 - the "URI spec". (informational status)

The object returned by `URLopener().open(file)` will differ per protocol. All you know is that it has methods `read()`, `readline()`, `readlines()`, `fileno()`, `close()` and `info()`. The `read*()`, `fileno()` and `close()` methods work like those of open files. The `info()` method returns a `mimetools.Message` object which can be used to query various info about the object, if available. (`mimetools.Message` objects are queried with the `getheader()` method.)

### 21.1 addbase

None

**Signature:**

`urllib.addbase(<object> fp)`

- **<object> fp:**

## 21.2 addbase.close

None

**Signature:**

```
[addbase].close()
```

## 21.3 addclosehook

None

**Signature:**

```
urllib.addclosehook(<object> fp,  
                   <object> closehook)
```

- **<object> fp:**
- **<object> closehook:**

## 21.4 addclosehook.close

None

**Signature:**

```
[addclosehook].close()
```

## 21.5 addinfo

None

**Signature:**

```
urllib.addinfo(<object> fp,  
              <object> headers)
```

- **<object> fp:**
- **<object> headers:**

## 21.6 addinfo.close

None

**Signature:**

```
[addinfo].close()
```

## 21.7 addinfo.info

None

**Signature:**

```
[addinfo].info()
```

## 21.8 addinfourl

None

**Signature:**

```
urllib.addinfourl(<object> fp,  
                  <object> headers,  
                  <object> url,  
                  <object> code=None)
```

- **<object> fp:**
- **<object> headers:**
- **<object> url:**
- **<object> code (optional):** If omitted, defaults to None.

## 21.9 addinfourl.close

None

**Signature:**

```
[addinfourl].close()
```

## 21.10 addinfourl.getcode

None

**Signature:**

```
[addinfourl].getcode()
```

## 21.11 addinfourl.geturl

None

**Signature:**

```
[addinfourl].geturl()
```

## 21.12 addinfourl.info

None

**Signature:**

```
[addinfourl].info()
```

## 21.13 basejoin

Join a base URL and a possibly relative URL to form an absolute interpretation of the latter.

**Signature:**

```
urllib.basejoin(<object> base,  
                <object> url,  
                <object> allow_fragments=True)
```

- **<object> base:**
- **<object> url:**
- **<object> allow\_fragments (optional):** If omitted, defaults to True.

## 21.14 ContentTooShortError

None

**Signature:**

```
urllib.ContentTooShortError(<object> message,  
                             <object> content)
```

- <object> message:
- <object> content:

## 21.15 FancyURLopener

None

**Signature:**

```
urllib.FancyURLopener()
```

## 21.16 FancyURLopener.addheader

Add a header to be used by the HTTP interface only e.g. `u.addheader('Accept', 'sound/basic')`

**Signature:**

```
[FancyURLopener].addheader()
```

## 21.17 FancyURLopener.cleanup

None

**Signature:**

```
[FancyURLopener].cleanup()
```

## 21.18 FancyURLopener.close

None

**Signature:**

```
[FancyURLopener].close()
```

## 21.19 FancyURLopener.get\_user\_passwd

None

**Signature:**

```
[FancyURLopener].get_user_passwd(<object> host,  
                                   <object> realm,  
                                   <object> clear_cache=0)
```

- **<object> host:**
- **<object> realm:**
- **<object> clear\_cache (optional):** If omitted, defaults to 0.

## 21.20 FancyURLopener.http\_error

Handle http errors. Derived class can override this, or provide specific handlers named `http_error_DDD` where DDD is the 3-digit error code.

**Signature:**

```
[FancyURLopener].http_error(<object> url,  
                             <object> fp,  
                             <object> errcode,  
                             <object> errmsg,  
                             <object> headers,  
                             <object> data=None)
```

- **<object> url:**



- `<object> fp`:
- `<object> errcode`:
- `<object> errmsg`:
- `<object> headers`:
- `<object> data` (optional): If omitted, defaults to `None`.

## 21.21 `FancyURLopener.http_error_301`

Error 301 – also relocated (permanently).

**Signature:**

```
[FancyURLopener].http_error_301(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- `<object> url`:
- `<object> fp`:
- `<object> errcode`:
- `<object> errmsg`:
- `<object> headers`:
- `<object> data` (optional): If omitted, defaults to `None`.

## 21.22 `FancyURLopener.http_error_302`

Error 302 – relocated (temporarily).

**Signature:**

```
[FancyURLopener].http_error_302(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- <object> url:
- <object> fp:
- <object> errcode:
- <object> errmsg:
- <object> headers:
- <object> data (optional): If omitted, defaults to None.

## 21.23 FancyURLopener.http\_error\_303

Error 303 – also relocated (essentially identical to 302).

Signature:

```
[FancyURLopener].http_error_303(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- <object> url:
- <object> fp:
- <object> errcode:
- <object> errmsg:
- <object> headers:
- <object> data (optional): If omitted, defaults to None.

## 21.24 FancyURLopener.http\_error\_307

Error 307 – relocated, but turn POST into error.

**Signature:**

```
[FancyURLopener].http_error_307(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- <object> url:
- <object> fp:
- <object> errcode:
- <object> errmsg:
- <object> headers:
- <object> data (optional): If omitted, defaults to None.

## 21.25 FancyURLopener.http\_error\_401

Error 401 – authentication required. This function supports Basic authentication only.

**Signature:**

```
[FancyURLopener].http_error_401(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- <object> url:

- `<object> fp`:
- `<object> errcode`:
- `<object> errmsg`:
- `<object> headers`:
- `<object> data` (optional): If omitted, defaults to `None`.

## 21.26 FancyURLopener.http\_error\_407

Error 407 – proxy authentication required. This function supports Basic authentication only.

**Signature:**

```
[FancyURLopener].http_error_407(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers,  
                                <object> data=None)
```

- `<object> url`:
- `<object> fp`:
- `<object> errcode`:
- `<object> errmsg`:
- `<object> headers`:
- `<object> data` (optional): If omitted, defaults to `None`.

## 21.27 FancyURLopener.http\_error\_default

Default error handling – don't raise an exception.

**Signature:**

```
[FancyURLopener].http_error_default(<object> url,  
                                     <object> fp,  
                                     <object> errcode,  
                                     <object> errmsg,  
                                     <object> headers)
```

- <object> url:
- <object> fp:
- <object> errcode:
- <object> errmsg:
- <object> headers:

## 21.28 FancyURLopener.open

Use `URLopener().open(file)` instead of `open(file, 'r')`.

**Signature:**

```
[FancyURLopener].open(<object> fullurl,  
                      <object> data=None)
```

- <object> fullurl:
- <object> data (optional): If omitted, defaults to None.

## 21.29 FancyURLopener.open\_data

Use "data" URL.

**Signature:**

```
[FancyURLopener].open_data(<object> url,  
                           <object> data=None)
```

- **<object> url:**
- **<object> data (optional):** If omitted, defaults to None.

### 21.30 FancyURLopener.open\_file

Use local file or FTP depending on form of URL.

**Signature:**

```
[FancyURLopener].open_file(<object> url)
```

- **<object> url:**

### 21.31 FancyURLopener.open\_ftp

Use FTP protocol.

**Signature:**

```
[FancyURLopener].open_ftp(<object> url)
```

- **<object> url:**

### 21.32 FancyURLopener.open\_http

Use HTTP protocol.

**Signature:**

```
[FancyURLopener].open_http(<object> url,  
                           <object> data=None)
```

- **<object> url:**
- **<object> data (optional):** If omitted, defaults to None.

### 21.33 FancyURLopener.open\_https

Use HTTPS protocol.

**Signature:**

```
[FancyURLopener].open_https(<object> url,  
                             <object> data=None)
```

- **<object> url:**
- **<object> data (optional):** If omitted, defaults to None.

### 21.34 FancyURLopener.open\_local\_file

Use local file.

**Signature:**

```
[FancyURLopener].open_local_file(<object> url)
```

- **<object> url:**

### 21.35 FancyURLopener.open\_unknown

Overridable interface to open unknown URL type.

**Signature:**

```
[FancyURLopener].open_unknown(<object> fullurl,  
                              <object> data=None)
```

- **<object> fullurl:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.36 FancyURLopener.open\_unknown\_proxy

Overridable interface to open unknown URL type.

**Signature:**

```
[FancyURLopener].open_unknown_proxy(<object> proxy,  
                                     <object> fullurl,  
                                     <object> data=None)
```

- <object> proxy:
- <object> fullurl:
- <object> data (optional): If omitted, defaults to None.

## 21.37 FancyURLopener.prompt\_user\_passwd

Override this in a GUI environment!

**Signature:**

```
[FancyURLopener].prompt_user_passwd(<object> host,  
                                     <object> realm)
```

- <object> host:
- <object> realm:

## 21.38 FancyURLopener.redirect\_internal

None

**Signature:**

```
[FancyURLopener].redirect_internal(<object> url,  
                                   <object> fp,  
                                   <object> errcode,  
                                   <object> errmsg,  
                                   <object> headers,  
                                   <object> data)
```



- `<object> url:`
- `<object> fp:`
- `<object> errcode:`
- `<object> errmsg:`
- `<object> headers:`
- `<object> data:`

### 21.39 FancyURLopener.retrieve

`retrieve(url)` returns (filename, headers) for a local object or (tempfilename, headers) for a remote object.

#### Signature:

```
[FancyURLopener].retrieve(<object> url,  
                           <object> filename=None,  
                           <object> reporthook=None,  
                           <object> data=None)
```

- `<object> url:`
- `<object> filename` (optional): If omitted, defaults to None.
- `<object> reporthook` (optional): If omitted, defaults to None.
- `<object> data` (optional): If omitted, defaults to None.

### 21.40 FancyURLopener.retry\_http\_basic\_auth

None

#### Signature:

```
[FancyURLopener].retry_http_basic_auth(<object> url,  
                                         <object> realm,  
                                         <object> data=None)
```

- **<object> url:**
- **<object> realm:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.41 FancyURLopener.retry\_https\_basic\_auth

None

**Signature:**

```
[FancyURLopener].retry_https_basic_auth(<object> url,  
                                         <object> realm,  
                                         <object> data=None)
```

- **<object> url:**
- **<object> realm:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.42 FancyURLopener.retry\_proxy\_http\_basic\_auth

None

**Signature:**

```
[FancyURLopener].retry_proxy_http_basic_auth(<object> url,  
                                              <object> realm,  
                                              <object> data=None)
```

- **<object> url:**
- **<object> realm:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.43 FancyURLopener.retry\_proxy\_https\_basic\_auth

None

**Signature:**

```
[FancyURLopener].retry_proxy_https_basic_auth(<object> url,  
                                              <object> realm,  
                                              <object> data=None)
```

- <object> url:
- <object> realm:
- <object> data (optional): If omitted, defaults to None.

## 21.44 ftpliberrors

Return the set of errors raised by the FTP class.

**Signature:**

```
urllib.ftpliberrors()
```

## 21.45 ftplibwrapper

None

**Signature:**

```
urllib.ftplibwrapper(<object> user,  
                    <object> passwd,  
                    <object> host,  
                    <object> port,  
                    <object> dirs,  
                    <object> timeout=<object object at 0x16b488>)
```

- <object> user:
- <object> passwd:

- `<object> host:`
- `<object> port:`
- `<object> dirs:`
- `<object> timeout` (optional): If omitted, defaults to `<object>` object at `0x16b488>`.

## 21.46 `ftpwrapper.close`

None

**Signature:**

```
[ftpwrapper].close()
```

## 21.47 `ftpwrapper.endtransfer`

None

**Signature:**

```
[ftpwrapper].endtransfer()
```

## 21.48 `ftpwrapper.init`

None

**Signature:**

```
[ftpwrapper].init()
```

## 21.49 ftpwrapper.retrfile

None

**Signature:**

```
[ftpwrapper].retrfile(<object> file,  
                      <object> type)
```

- <object> file:
- <object> type:

## 21.50 getproxies

None

**Signature:**

```
urllib.getproxies()
```

## 21.51 getproxies\_environment

Return a dictionary of scheme -> proxy server URL mappings.

Scan the environment for variables named <scheme>\_proxy; this seems to be the standard convention. If you need a different way, you can pass a proxies dictionary to the [Fancy]URLopener constructor.

**Signature:**

```
urllib.getproxies_environment()
```

## 21.52 getproxies\_macosx\_sysconf

Return a dictionary of scheme -> proxy server URL mappings.

This function uses the MacOSX framework SystemConfiguration to fetch the proxy information.

**Signature:**

```
urllib.getproxies_macosx_sysconf()
```

## 21.53 localhost

Return the IP address of the magic hostname 'localhost'.

**Signature:**

```
urllib.localhost()
```

## 21.54 main

None

**Signature:**

```
urllib.main()
```

## 21.55 noheaders

Return an empty `mimetools.Message` object.

**Signature:**

```
urllib.noheaders()
```

## 21.56 pathname2url

OS-specific conversion from a file system path to a relative URL of the 'file' scheme; not recommended for general use.

**Signature:**

```
urllib.pathname2url(<object> pathname)
```

- **<object> pathname:**

## 21.57 proxy\_bypass

None

**Signature:**

```
urllib.proxy_bypass(<object> host)
```

- <object> host:

## 21.58 proxy\_bypass\_environment

Test if proxies should not be used for a particular host.

Checks the environment for a variable named `no_proxy`, which should be a list of DNS suffixes separated by commas, or `'*'` for all hosts.

**Signature:**

```
urllib.proxy_bypass_environment(<object> host)
```

- <object> host:

## 21.59 proxy\_bypass\_macosx\_sysconf

Return True iff this host shouldn't be accessed using a proxy

This function uses the MacOSX framework `SystemConfiguration` to fetch the proxy information.

**Signature:**

```
urllib.proxy_bypass_macosx_sysconf(<object> host)
```

- <object> host:

## 21.60 quote

`quote('abc def') -> 'abc%20def'`

Each part of a URL, e.g. the path info, the query, etc., has a different set of reserved characters that must be quoted.

RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax lists the following reserved characters.

reserved = ";" — "/" — "?" — ":" — "@" — "&" — "=" — "+" — "\$" — ","

Each of these characters is reserved in some component of a URL, but not necessarily in all of them.

By default, the quote function is intended for quoting the path section of a URL. Thus, it will not encode '/'. This character is reserved, but in typical usage the quote function is being called on a path where the existing slash characters are used as reserved characters.

### Signature:

```
urllib.quote(<object> s,
             <object> safe=)
```

- **<object> s:**
- **<object> safe (optional):** If omitted, defaults to /.

## 21.61 quote\_plus

Quote the query fragment of a URL; replacing ' ' with '+'

### Signature:

```
urllib.quote_plus(<object> s,
                  <object> safe=)
```

- **<object> s:**
- **<object> safe (optional):** If omitted, defaults to .



## 21.62 reporthook

None

**Signature:**

```
urllib.reporthook(<object> blocknum,  
                  <object> blocksize,  
                  <object> totalsize)
```

- <object> blocknum:
- <object> blocksize:
- <object> totalsize:

## 21.63 splitattr

`splitattr('/path;attr1=value1;attr2=value2;...') -> '/path', ['attr1=value1', 'attr2=value2', ...]`.

**Signature:**

```
urllib.splitattr(<object> url)
```

- <object> url:

## 21.64 splithost

`splithost('//host[:port]/path') -> 'host[:port]', '/path'.`

**Signature:**

```
urllib.splithost(<object> url)
```

- <object> url:

## 21.65 splitnport

Split host and port, returning numeric port. Return given default port if no ':' found; defaults to -1. Return numerical port if a valid number are found after ':'. Return None if ':' but not a valid number.

### Signature:

```
urllib.splitnport(<object> host,  
                 <object> defport=-1)
```

- **<object> host:**
- **<object> defport (optional):** If omitted, defaults to -1.

## 21.66 splitpasswd

splitpasswd('user:passwd') -> 'user', 'passwd'.

### Signature:

```
urllib.splitpasswd(<object> user)
```

- **<object> user:**

## 21.67 splitport

splitport('host:port') -> 'host', 'port'.

### Signature:

```
urllib.splitport(<object> host)
```

- **<object> host:**

## 21.68 splitquery

`splitquery('/path?query') -> '/path', 'query'.`

**Signature:**

```
urllib.splitquery(<object> url)
```

- **<object> url:**

## 21.69 splittag

`splittag('/path#tag') -> '/path', 'tag'.`

**Signature:**

```
urllib.splittag(<object> url)
```

- **<object> url:**

## 21.70 splittype

`splittype('type:opaquestring') -> 'type', 'opaquestring'.`

**Signature:**

```
urllib.splittype(<object> url)
```

- **<object> url:**

## 21.71 splituser

`splituser('user[:passwd]@host[:port]') -> 'user[:passwd]', 'host[:port]'.`

**Signature:**

```
urllib.splituser(<object> host)
```

- **<object> host:**

## 21.72 splitvalue

`splitvalue('attr=value') -> 'attr', 'value'.`

**Signature:**

```
urllib.splitvalue(<object> attr)
```

- `<object> attr`:

## 21.73 test

None

**Signature:**

```
urllib.test(<object> args=[])
```

- `<object> args` (optional): If omitted, defaults to `[]`.

## 21.74 test1

None

**Signature:**

```
urllib.test1()
```

## 21.75 thishost

Return the IP address of the current host.

**Signature:**

```
urllib.thishost()
```

## 21.76 toBytes

`toBytes(u"URL") -> 'URL'`.

**Signature:**

```
urllib.toBytes(<object> url)
```

- `<object> url:`

## 21.77 unquote

`unquote('abc%20def') -> 'abc def'`.

**Signature:**

```
urllib.unquote(<object> s)
```

- `<object> s:`

## 21.78 unquote\_plus

`unquote('%7e/abc+def') -> ' /abc def'`

**Signature:**

```
urllib.unquote_plus(<object> s)
```

- `<object> s:`

## 21.79 unwrap

`unwrap('<URL:type://host/path>') -> 'type://host/path'`.

**Signature:**

```
urllib.unwrap(<object> url)
```

- `<object> url:`

## 21.80 url2pathname

OS-specific conversion from a relative URL of the 'file' scheme to a file system path; not recommended for general use.

**Signature:**

```
urllib.url2pathname(<object> pathname)
```

- **<object> pathname:**

## 21.81 urlcleanup

None

**Signature:**

```
urllib.urlcleanup()
```

## 21.82 urlencode

Encode a sequence of two-element tuples or dictionary into a URL query string.

If any values in the query arg are sequences and doseq is true, each sequence element is converted to a separate parameter.

If the query arg is a sequence of two-element tuples, the order of the parameters in the output will match the order of parameters in the input.

**Signature:**

```
urllib.urlencode(<object> query,  
                 <object> doseq=0)
```

- **<object> query:**
- **<object> doseq (optional):** If omitted, defaults to 0.

## 21.83 urlopen

Create a file-like object for the specified URL to read from.

**Signature:**

```
urllib.urlopen(<object> url,  
               <object> data=None,  
               <object> proxies=None)
```

- **<object> url:**
- **<object> data (optional):** If omitted, defaults to None.
- **<object> proxies (optional):** If omitted, defaults to None.

## 21.84 URLOpener

None

**Signature:**

```
urllib.URLOpener(<object> proxies=None)
```

- **<object> proxies (optional):** If omitted, defaults to None.

## 21.85 URLOpener.addheader

Add a header to be used by the HTTP interface only e.g. `u.addheader('Accept', 'sound/basic')`

**Signature:**

```
[URLOpener].addheader()
```

## 21.86 URLOpener.cleanup

None

**Signature:**

```
[URLOpener].cleanup()
```

## 21.87 URLOpener.close

None

**Signature:**

```
[URLOpener].close()
```

## 21.88 URLOpener.http\_error

Handle http errors. Derived class can override this, or provide specific handlers named `http_error_DDD` where DDD is the 3-digit error code.

**Signature:**

```
[URLOpener].http_error(<object> url,  
                        <object> fp,  
                        <object> errcode,  
                        <object> errmsg,  
                        <object> headers,  
                        <object> data=None)
```

- `<object> url`:
- `<object> fp`:
- `<object> errcode`:
- `<object> errmsg`:
- `<object> headers`:
- `<object> data` (optional): If omitted, defaults to None.

## 21.89 URLOpener.http\_error\_default

Default error handler: close the connection and raise IOError.

**Signature:**



```
[URLopener].http_error_default(<object> url,  
                                <object> fp,  
                                <object> errcode,  
                                <object> errmsg,  
                                <object> headers)
```

- <object> url:
- <object> fp:
- <object> errcode:
- <object> errmsg:
- <object> headers:

## 21.90 URLopener.open

Use `URLopener().open(file)` instead of `open(file, 'r')`.

**Signature:**

```
[URLopener].open(<object> fullurl,  
                 <object> data=None)
```

- <object> fullurl:
- <object> data (optional): If omitted, defaults to `None`.

## 21.91 URLopener.open\_data

Use "data" URL.

**Signature:**

```
[URLopener].open_data(<object> url,  
                      <object> data=None)
```

- <object> url:
- <object> data (optional): If omitted, defaults to `None`.

## 21.92 URLOpener.open\_file

Use local file or FTP depending on form of URL.

**Signature:**

```
[URLOpener].open_file(<object> url)
```

- <object> url:

## 21.93 URLOpener.open\_ftp

Use FTP protocol.

**Signature:**

```
[URLOpener].open_ftp(<object> url)
```

- <object> url:

## 21.94 URLOpener.open\_http

Use HTTP protocol.

**Signature:**

```
[URLOpener].open_http(<object> url,  
                      <object> data=None)
```

- <object> url:
- <object> data (optional): If omitted, defaults to None.

## 21.95 URLOpener.open\_https

Use HTTPS protocol.

**Signature:**

```
[URLopener].open_https(<object> url,  
                        <object> data=None)
```

- **<object> url:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.96 URLopener.open\_local\_file

Use local file.

**Signature:**

```
[URLopener].open_local_file(<object> url)
```

- **<object> url:**

## 21.97 URLopener.open\_unknown

Overridable interface to open unknown URL type.

**Signature:**

```
[URLopener].open_unknown(<object> fullurl,  
                          <object> data=None)
```

- **<object> fullurl:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.98 URLopener.open\_unknown\_proxy

Overridable interface to open unknown URL type.

**Signature:**

```
[URLopener].open_unknown_proxy(<object> proxy,  
                               <object> fullurl,  
                               <object> data=None)
```

- **<object> proxy:**
- **<object> fullurl:**
- **<object> data (optional):** If omitted, defaults to None.

## 21.99 URLOpener.retrieve

retrieve(url) returns (filename, headers) for a local object or (tempfilename, headers) for a remote object.

### Signature:

```
[URLOpener].retrieve(<object> url,  
                    <object> filename=None,  
                    <object> reporthook=None,  
                    <object> data=None)
```

- **<object> url:**
- **<object> filename (optional):** If omitted, defaults to None.
- **<object> reporthook (optional):** If omitted, defaults to None.
- **<object> data (optional):** If omitted, defaults to None.

## 21.100 urlretrieve

None

### Signature:

```
urllib.urlretrieve(<object> url,  
                  <object> filename=None,  
                  <object> reporthook=None,  
                  <object> data=None)
```

- **<object> url:**
- **<object> filename (optional):** If omitted, defaults to None.

- `<object> reporthook` (optional): If omitted, defaults to `None`.
- `<object> data` (optional): If omitted, defaults to `None`.

# Chapter 22

## xml.etree.ElementTree

### 22.1 Comment

None

**Signature:**

```
xml.etree.ElementTree.Comment(<object> text=None)
```

- <object> text (optional): If omitted, defaults to None.

### 22.2 dump

None

**Signature:**

```
xml.etree.ElementTree.dump(<object> elem)
```

- <object> elem:

### 22.3 Element

None

**Signature:**

```
xml.etree.ElementTree.Element(<object> tag,  
                               <object> attrib={})
```

- **<object> tag:**
- **<object> attrib (optional):** If omitted, defaults to {}.

## 22.4 ElementTree

None

**Signature:**

```
xml.etree.ElementTree.ElementTree(<object> element=None,  
                                   <object> file=None)
```

- **<object> element (optional):** If omitted, defaults to None.
- **<object> file (optional):** If omitted, defaults to None.

## 22.5 ElementTree.find

None

**Signature:**

```
[ElementTree].find(<object> path)
```

- **<object> path:**

## 22.6 ElementTree.findall

None

**Signature:**

```
[ElementTree].findall(<object> path)
```

- **<object> path:**

## 22.7 ElementTree.findtext

None

**Signature:**

```
[ElementTree].findtext(<object> path,  
                        <object> default=None)
```

- <object> path:
- <object> default (optional): If omitted, defaults to None.

## 22.8 ElementTree.getiterator

None

**Signature:**

```
[ElementTree].getiterator(<object> tag=None)
```

- <object> tag (optional): If omitted, defaults to None.

## 22.9 ElementTree.getroot

None

**Signature:**

```
[ElementTree].getroot()
```

## 22.10 ElementTree.parse

None

**Signature:**

```
[ElementTree].parse(<object> source,  
                    <object> parser=None)
```



- **<object> source:**
- **<object> parser (optional):** If omitted, defaults to None.

## 22.11 ElementTree.write

None

**Signature:**

```
[ElementTree].write(<object> file,  
                    <object> encoding=us-ascii)
```

- **<object> file:**
- **<object> encoding (optional):** If omitted, defaults to us-ascii.

## 22.12 fixtag

None

**Signature:**

```
xml.etree.ElementTree.fixtag(<object> tag,  
                             <object> namespaces)
```

- **<object> tag:**
- **<object> namespaces:**

## 22.13 fromstring

None

**Signature:**

```
xml.etree.ElementTree.fromstring(<object> text)
```

- **<object> text:**

## 22.14 iselement

None

**Signature:**

```
xml.etree.ElementTree.iselement(<object> element)
```

- <object> element:

## 22.15 iterparse

None

**Signature:**

```
xml.etree.ElementTree.iterparse(<object> source,  
                                <object> events=None)
```

- <object> source:
- <object> events (optional): If omitted, defaults to None.

## 22.16 iterparse.next

None

**Signature:**

```
[iterparse].next()
```

## 22.17 parse

None

**Signature:**

```
xml.etree.ElementTree.parse(<object> source,  
                             <object> parser=None)
```

- **<object> source:**
- **<object> parser (optional):** If omitted, defaults to None.

## 22.18 PI

None

**Signature:**

```
xml.etree.ElementTree.PI(<object> target,  
                          <object> text=None)
```

- **<object> target:**
- **<object> text (optional):** If omitted, defaults to None.

## 22.19 ProcessingInstruction

None

**Signature:**

```
xml.etree.ElementTree.ProcessingInstruction(<object> target,  
                                             <object> text=None)
```

- **<object> target:**
- **<object> text (optional):** If omitted, defaults to None.

## 22.20 QName

None

**Signature:**

```
xml.etree.ElementTree.QName(<object> text_or_uri,  
                             <object> tag=None)
```

- **<object> text\_or\_uri:**
- **<object> tag (optional):** If omitted, defaults to None.

## 22.21 SubElement

None

**Signature:**

```
xml.etree.ElementTree.SubElement(<object> parent,  
                                  <object> tag,  
                                  <object> attrib={})
```

- <object> parent:
- <object> tag:
- <object> attrib (optional): If omitted, defaults to {}.

## 22.22 tostring

None

**Signature:**

```
xml.etree.ElementTree.tostring(<object> element,  
                                <object> encoding=None)
```

- <object> element:
- <object> encoding (optional): If omitted, defaults to None.

## 22.23 TreeBuilder

None

**Signature:**

```
xml.etree.ElementTree.TreeBuilder(<object> element_factory=None)
```

- <object> element factory (optional): If omitted, defaults to None.

## 22.24 TreeBuilder.close

None

**Signature:**

```
[TreeBuilder].close()
```

## 22.25 TreeBuilder.data

None

**Signature:**

```
[TreeBuilder].data(<object> data)
```

- <object> data:

## 22.26 TreeBuilder.end

None

**Signature:**

```
[TreeBuilder].end(<object> tag)
```

- <object> tag:

## 22.27 TreeBuilder.start

None

**Signature:**

```
[TreeBuilder].start(<object> tag,  
                    <object> attrs)
```

- <object> tag:
- <object> attrs:

## 22.28 XML

None

**Signature:**

```
xml.etree.ElementTree.XML(<object> text)
```

- <object> text:

## 22.29 XMLID

None

**Signature:**

```
xml.etree.ElementTree.XMLID(<object> text)
```

- <object> text:

## 22.30 XMLParser

None

**Signature:**

```
xml.etree.ElementTree.XMLParser(<object> html=0,  
                                <object> target=None)
```

- <object> html (optional): If omitted, defaults to 0.
- <object> target (optional): If omitted, defaults to None.

## 22.31 XMLParser.close

None

**Signature:**

```
[XMLParser].close()
```

## 22.32 XMLParser.doctype

None

**Signature:**

```
[XMLParser].doctype(<object> name,  
                    <object> pubid,  
                    <object> system)
```

- <object> name:
- <object> pubid:
- <object> system:

## 22.33 XMLParser.feed

None

**Signature:**

```
[XMLParser].feed(<object> data)
```

- <object> data:

## 22.34 XMLTreeBuilder

None

**Signature:**

```
xml.etree.ElementTree.XMLTreeBuilder(<object> html=0,  
                                       <object> target=None)
```

- <object> html (optional): If omitted, defaults to 0.
- <object> target (optional): If omitted, defaults to None.

## 22.35 XMLTreeBuilder.close

None

**Signature:**

```
[XMLTreeBuilder].close()
```

## 22.36 XMLTreeBuilder.doctype

None

**Signature:**

```
[XMLTreeBuilder].doctype(<object> name,  
                           <object> pubid,  
                           <object> system)
```

- <object> name:
- <object> pubid:
- <object> system:

## 22.37 XMLTreeBuilder.feed

None

**Signature:**

```
[XMLTreeBuilder].feed(<object> data)
```

- <object> data:

## 22.38 Additional Libraries

Picalo comes with several other libraries that, while not core to its function, are generally useful in working with data. In addition, Python brings a vast number of additional libraries to bear. These libraries are described below.



### 22.38.1 GUID

A globally-unique identifier (GUID) is a sequence of characters that is unique across space and time. GUIDs can be created independently, and yet they still remain unique from one another. This is accomplished by combining the current system time, the system network address, and several random characters. GUIDs are *not perfectly* unique, but they are close enough to make them useful.

The primary use of a GUID is in the creation of a database key. Consider the traditional methods of creating database keys: an autoincrementing column or a sequential number column. For the automatically incrementing column, you don't know the key of the record you are inserting until after you insert it (thus, it requires a separate query statement). For the sequential number column, you first have to select the maximum value from the column, increment, and use the new number. Both have many points of failure. What if two programs select the maximum value from the same table at the same time (they both get the same maximum value)? What if two independent tables are joined together (their keys clash)? Changing of key values is also not a good option: keys often have many other tables and foreign keys dependent upon their values.

GUIDs solve all of this because they can be created without any database access and (almost) guarantee uniqueness across all databases, tables, space, and time. Stated differently, millions of potential GUIDs are being created every hour of every day; we just need to create them.

GUIDs also have other advantages. Since they encode time and IP, you always know when and where a database record was created.

The GUID module provides a **generate** method to create GUIDs, and it provides methods to extract the creation time, IP, and random number from existing GUIDs.

Picalo's particular GUID module keeps track of the last 100 GUIDs generated to ensure uniqueness, and it is unique to the millisecond. It uses eight random characters (essentially, the positive range of Python's integer). Each GUID is exactly 40 characters in length.

```
1 from picalo.lib import GUID
2
3 guid = GUID.generate()
4 print guid
5 print GUID.extract_time(guid)
6 print GUID.extract_ip(guid)
7 print GUID.extract_random(guid)
```

### 22.38.2 Statistics

The stats.py module included with Picalo was written by Gary Strangman. Stats.py is used in many other Python applications and libraries, such as the popular Numeric/Numarray package. As such, you can be sure it is tested and mature.

Python's built in functions provide basic statistical functions, such as len, max, and min. These functions don't require the stats.py module. More advanced routines, such as regression, t tests, and chi-square tests are supported by stats. See the stats.py file directly for information about how to use the routines in this package. A short example follows:

```

1 from picalo.lib import stats
2
3 # create a sample table
4 table = Table(['Name', 'Sales'], [
5     ['Randy', 100000 ],
6     ['Ted', 40000 ],
7     ['Suzy', 160000 ],
8     ['Vijay', 240000 ],
9     ['Stan', 56000 ]
10 ])
11
12 # get descriptive statistics
13 print stats.mean(table['Sales'])
14 # prints 119200.0
15 print stats.stdev(table['Sales'])
16 # prints 81995.12
17 print len(table['Sales'])
18 # prints 2
19 print max(table['Sales'])
20 # prints 240000

```

### 22.38.3 Graphs and Plots

The Windows and Mac OS X binary distributions of Picalo include the excellent *matplotlib* graphing library. For those installing the source distribution, download the package from <http://matplotlib.sourceforge.net/>.

Picalo's GUI includes many menu options for the creation of graphs of many types. These are only the beginnings of what can be done with *matplotlib*. See the documentation on this library for further instructions.

### 22.38.4 Python Libraries

A casual reviewer of Picalo may not see the power of extending the Python language and may judge Picalo as a limited software package. Python alone

is a very powerful data and text analysis platform – Picalo only adds data analysis functions not already implemented in Python.

Python comes with a rich library of modules you'll likely find useful. These modules are documented in the standard Python documentation available at <http://www.python.org/doc/>. In addition, thousands upon thousands of user-written libraries are available for download on the Internet. A simple search of the World Wide Web should reveal many examples of routines you need to run. This sections lists specific Python modules that are especially interesting to analysts. It is only an example of the libraries available for your use.

- **core functions:** Python includes many core functions that are globally available anywhere in Picalo. These functions include methods such as `abs` (absolute value), `cmp` (compare), `dict` (create a dictionary), `float` (convert to real), `int` (convert to integer), `str` (convert to string), `len` (length of a list or Picalo table), `max` (maximum value in a list/column), `min` (minimum value in a list/column), `range` (create a list of numbers), `round` (round a number), `sum` (total a list of numbers), `type` (return the type of an object), and many other all-around useful methods.
- **re:** This module handles regular expressions: a powerful language that searches for patterns in text. While regular expressions take a few hours to learn, the payoff in searching through text is enormous. For example, you can import printed reports (and remove the junk like page headers and footers) easily with `re`. Standalone tools to do this often cost thousands. *If you need to search through a lot of text or parse reports, you won't get a more powerful toolkit than regular expressions.* The regular expression language is decades old and shows the marks of experience, maturity, and power.
- **string:** The string module contains methods that act on string values, such as converting to upper/lower case, etc.
- **random:** This module creates random numbers within ranges you specify. It comes in handy when selecting random records for investigation.
- **math:** The math module contains advanced mathematical functions, such as exponential calculations, square roots, etc.

- **gzip**: This module implements the compression algorithm behind zip programs such as WinZip. It can compress and uncompress text files on the fly, as if they were regular (uncompressed) text files. This module is useful if you are working with *really* big files: you can store the files on your disk in compressed form, and your scripts can uncompress them for analysis on the fly.
- **xml**: Python contains many different xml-based libraries, all beginning with xml.\*. These libraries allow you to read and write xml files using the *dom* or *sax* interfaces. In particular, the xml.dom.minidom package makes reading and writing xml documents easy.

# Appendix A

## Creating Detectlets

Detectlets are one of the most exciting parts of the Picalo framework. Detectlets allow you to codify your work in an established, robust framework, and they allow others to benefit from your genius. Detectlets give you a way to contribute back a little bit to a community that has worked hard to provide this tool.

You are even free to charge for the use of your Detectlets, should you choose to do so. I expect that some users will give their Detectlets away for free and others will sell them commercially. Either way, you'll benefit the community and make us all more productive.

First and foremost, the goal of a Detectlet is to walk an unskilled user through your analysis algorithm. Detectlets should be simple, simple, simple. They should contain verbose wording to explain why the Detectlet does what it does, where it should be used, its background, the assumptions it makes, and as much other information as you can give. Try to put yourself in the place of a person learning your algorithm for the first time. Don't talk down to the user, but certainly try to be descriptive. Help the user learn what you know (or at least, help the user learn how to utilize what you know).

Detectlets should be written to the domain rather than to the abstract analysis being done. Remember that many users won't see the potential of many basic routines. Picalo already does many abstract analysis techniques, such as stratification, regular expressions, loading data, and so forth. Detectlets should be specific to a single use.

For example, comparing columns from two tables is a basic analysis routine. This analysis can be used to compare employee addresses to vendor addresses, compare student assignments to one another, or do a hundred

other things. Each of these things should turn into a Detectlet, even though the analysis is essentially the same. In other words, do not create a Detectlet named “Compare Columns From Tables”; create several Detectlets named “Find Phantom Vendors By Comparing Employee Addresses to Vendor Addresses”, “Discover Potential Cheating By Comparing Student Assignments With On Another”, and so forth.

Detectlets assume that the user has already loaded one or more tables into memory. Each Detectlet should take these table(s) as input. After the analysis is completed, each Detectlet should return a Table or TableList containing the results.

## A.1 The Detectlet Process

Each Detectlet starts with a description of its purpose, domain, background, and assumptions. The opening page of the wizard also (hopefully) shows the user example data. Most users want to see example data to know what their input data should look like.

The next few pages of the wizard collect input data, settings, and user preferences for the analysis. The user should be able to select the table(s) to be analyzed, the individual columns of interest, and any other settings important to the analysis. You do not need to expose every nuance of your analysis to the user. Instead, you may want to make a few assumptions or create multiple Detectlets with the same routine but with different settings. Most users will understand multiple Detectlets better than a single Detectlet with lots of settings.

The input data should generally be in Picalo Table or TableList format. It is important that Picalo routines always input tables and output tables. This allows routines to be chained together to create even more complex and comprehensive routines.

The input pages of the wizard are defined by your Detectlet XML. When all data is collected (according to your Detectlet XML), the wizard will present a final page asking the user for a variable name to store the results in.

When the user clicks the *Finish* button, Picalo will call your routine with the data inputted by the user. If your routine throws an exception, the user will be notified and will be allowed to adjust the input settings.

When your analysis finishes successfully, the results table (returned by

your routine) is displayed in Picalo. A popup window displays on top of this window to teach the user how to interpret the results table. The user is free to close this window or study its contents as well as the table.

## A.2 Detectlet Anatomy

A Detectlet is a Python file with a special structure. To install a detectlet, simply place it in the “detectlets” directory (or one of its subdirectories) – Picalo can do this for you if you select Detectlets — Install Detectlet Library from the menu. Place only one Detectlet per source file.

The name of the Detectlet file is used to populate the Detectlet’s menu in Picalo. Use capital letters to signify the words of your Detectlet. A Detectlet named “Find Phantom Vendors By Comparing Employee Addresses to Vendor Addresses” should be contained in a file named “FindPhantomVendorsByComparingEmployeeAddressestoVendorAddresses.py”.

Each Detectlet file has four members: the `version`, the `wizard` variable, the `run` analysis function, and the `example_input` function. These members are described in the next subsections.

If you develop your Detectlet in the Picalo editor, you can test run your Detectlet by selecting Script — Run Script As Detectlet. This will run the Detectlet directly from the editor (without direct installation into Picalo) so you can debug and test it.

Picalo reloads Detectlets *every* time they are run, so you don’t need to restart Picalo every time you modify one during development. Each subsequent run of the wizard will reload any new changes from your files.

Since Picalo comes with several example Detectlets, be sure to open these files in the editor. These files give concrete examples of the requirements set forth in the next sections.

## A.3 Detectlet Version

As the standards may change over time in the way detectlets run, you must provide a “DETECTLET\_STANDARD” global variable within your detectlet file. This lets Picalo know what standard your detectlet conforms to.

The current standard is version 1.0. Be sure to include the following code near the top of your detectlet file:

```
1 DETECTLET_STANDARD = 1.0
```

## A.4 The wizard variable

The **wizard** variable is a multiline string that contains a short XML document. The purpose of the XML document is to outline the pages of the wizard. The purpose of the wizard is to gather information from the user about the parameters required to run the function.

The XML document has the following format:

```

1 <detectlet>
2   <page>
3     Instruction text for the user can go anywhere within the page tag.
4     <parameter type="parameter type" variable="function parameter name"/>
5     ... (other parameters for this page)
6   </page>
7   <page>
8     ... (additional parameters for the next page)
9   </page>
10  ... (additional pages for the wizard)
11 </detectlet>

```

As seen in the above XML, the document tells Picalo how many pages to show the user and what parameters to ask for on each page. It also contains instruction text so the user can understand what is needed. The following parameter types are supported:

**Integer Numbers** The integer number parameter asks the user to input an integer. The control supports minimum and maximum enforcement, a default value (shown when the control is first displayed), and a regular-expression-based mask.

```

1 <parameter type="int"
2     variable="varname"
3     min="0"
4     max="5"
5     default="1"
6     mask="regex"/>

```

**Floating-Point Numbers** The floating-point number parameter asks the user to input a float. The control supports minimum and maximum enforcement, a default value (shown when the control is first displayed), and a regular-expression-based mask.

```

1 <parameter type="float"
2     variable="varname"
3     min="0.0"
4     max="5.0"
5     default="1.0"
6     mask="regex"/>

```



**Strings** The string parameter asks the user to input a string of any type or length. The control a regular-expression-based mask, which gives total control over what the user enters into the control.

```
1 <parameter type="string"
2           variable="varname"
3           mask="regex"/>
```

**Database Connections** The database parameter shows the user the available databases in memory (previously created or loaded into Picalo). The database parameter returns an actual reference to a table in memory.

```
1 <parameter type="database"
2           variable="varname" />
```

**Tables** The table parameter shows the user the available tables in memory (previously created or loaded into Picalo). The table parameter supports an optional "multiple" attribute that allows multiple tables to be selected. The table parameter returns an actual reference to a table in memory or a list of tables in memory if multiple is true.

```
1 <parameter type="table"
2           variable="varname"
3           multiple="true"/>
```

**Columns** The column parameter allows the user to select a column from a table in memory. The column parameter type must be linked to a table parameter shown elsewhere in the wizard. Whenever the table parameter is changed, this parameter type will reload the list box with the columns from that table. The column parameter supports an optional "multiple" attribute that allows multiple columns to be selected. The column parameter returns the name of the selected column (as a string) or a list of strings if multiple is true.

```
1 <parameter type="column"
2           variable="varname"
3           table="tablevarname"
4           multiple="true"/>
```

**List Boxes** The list parameter shows a list box of options for the user to select from. One or more "option" child elements should be provided to give the lists available values. The optional "value" attribute allows options to display text one way and set the parameter value another (such as displaying "One" but sending "1" into the function). The choice parameter supports an optional "multiple" attribute that allows multiple choices to be selected. The list parameter always returns a string value (which you should convert to another type if needed).

```
1 <parameter type="list" variable="varname" multiple="true">
2   <option value="value">text</option>
3   ... (more options)
4 </parameter>
```

**Choice Boxes** The choice parameter shows a drop-down choice box of options for the user to select from. It does not support multiple selection of options. Otherwise, it is identical to the list parameter type.

```
1 <parameter type="choice" variable="varname">
2   <option value="value">text</option>
3   ... (more options)
4 </parameter>
```

## A.5 Analysis Function

The analysis function is the heart of your routine. Once the wizard gathers all the input data, Picalo will call your analysis function. Alternatively, your analysis function should support script-based use.

The analysis function must be named `run()` and can take any number of parameters. Normally, the `run` function takes one or two tables, one or two columns within those tables, and a few other settings.

The parameters of the function should be named exactly as they are given in the “variable” attribute of the wizard XML. Picalo will match these names when it calls your function. In other words, the wizard `parameter` elements must exactly match the parameters in your `run` function.

Parameters are typed as they come into your function. If the user selects a table in the wizard, the actual table object is sent to your script. If the user selects a column, the column name (as a string) is sent. Integers, floats, choices, and other input variables are sent via their types.

The function should contain documentation text (a string immediately following the function declaration) that describes the background of the routine, the input data, and the steps the analysis will perform. This text will be shown on the first page of the wizard. This is probably the most important part of the Detectlet because it explains where, how, and why your routine should be run.

The `run` function must return two items: A results table or table list, and a string. Upon completion of your function, Picalo will show your results table. If the string is nonempty, it will open a small frame containing the HTML string. The purpose of the string is to give follow-up information to

the user on how to interpret the results you present. You can have a standard string that returns from every call to your Detectlet, or you can custom build the string based upon the results your analysis finds. The window is a (simplified) HTML viewer, so you can format results using HTML if desired.

Following is a (relatively simple) example function:

```

1 def run(table, col):
2     '''This Detectlet retrieves a column from a table'''
3     results = Table([col])
4     for row in table:
5         results.append(row[col])
6     return results, '<html><body>The displayed table contains only the column you selected.</body></html>'

```

The content of the function is entirely up to you. Perform an analysis using Python and/or Picalo functions. If your function throws exceptions, the text of those exceptions will be shown to the user nicely (e.g. the program won't crash). You can use Python's **assert** statement to ensure the parameters came in correctly.

## A.6 Example Input

The `example_input` function takes no parameters and returns either a Picalo Table, TableList, or list of Tables containing example data. This gives the user an example of what the input data table should look like. You should also provide text that describes this input data in the function description.

To return a single Picalo Table (or TableList), simply return it from the function. The following example shows this most common format:

```

1 def example_input():
2     import StringIO # to emulate a file for load_csv
3     table = load_csv(StringIO.StringIO(csvdata))
4     table.set_type('Contract', int)
5     table.set_type('Amount', number)
6     return table
7
8 csvdata = '''\
9 Contract,Bidder,Item,Amount
10 1,BidderA,1.1,9908.01
11 1,BidderA,1.2,4147.38
12 '''

```

Some Detectlets require that multiple tables to perform their function. To return multiple tables, simply return a regular Python list or tuple of tables. The following example shows this format. Note that the function returns both `table`, `table2`. These two tables are not Picalo TableLists,

but instead are individual tables. Both will show in the Picalo GUI when the user clicks the example input button.

```

1 def example_input():
2     import StringIO # to emulate a file for load_csv
3     table = load_csv(StringIO.StringIO(csvdata))
4     table.set_type('Contract', int)
5     table.set_type('Amount', number)
6     table2 = load_csv(StringIO.StringIO(csvdata2))
7     return table, table2
8
9 csvdata = '''\
10 Contract,Bidder,Item,Amount
11 1,BidderA,1.1,9908.01
12 1,BidderA,1.2,4147.38
13 '''
14 csvdata2 = '''\
15 Some,Other,Table,Columns
16 1,2,3,7
17 4,5,6,8

```

The `example_input` function is optional. If it is not provided, the wizard won't have a button to open the example window.

# Appendix B

## Creating Plugins

(This section is for programmers who wish to extend Picalo.) Picalo is programmed using a plugin architecture that allows third parties to extend its behavior. These plugins can be released separate from Picalo (open source or for charge) or can be submitted to Conan Albrecht for inclusion in the main Picalo tree (subject to approval). Detectlets are the best example of a plugin for Picalo, and they come with the application. The entire Detectlet wizard is programmed as a plugin to the application.

When Picalo starts, it examines the picalo/tools directory for plugins. A plugin is defined as a subdirectory in picalo/tools that has an `__init__.py` file. The directory name must start with a regular letter (not underscore, dot, or a number).

For example, the Detectlet plugin has the following directory tree:

```
picalo
|
+-- tools
    |
    +-- Detectlets
        |
        +-- __init__.py
```

If you wished to create a new plugin called “MyCoolPlugin”, you would create the following:

```
picalo
|
```

```

+-- tools
    |
    +-- MyCoolPlugin
        |
        +-- __init__.py

```

The user, of course, never sees this directory tree. Instead, when Picalo starts, it discovers any plugins in `picalo/tools` and adds their menu items to the main menu of Picalo. The user sees a seamless application rather than a set of plugins.

The `__init__.py` file must define two methods and a variable. The variable, `PLUGIN_VERSION`, defines the plugin framework version. Currently only 1.0 is supported (this variable is actually ignored right now – it will become important when future changes are made to this framework).

The `initialize_plugin` method is called when the plugin starts (normally at application startup). It passes a reference to the main frame of the application (see `picalo/gui/MainFrame.py`). This reference is important when calling wx dialogs or when inspecting the greater Picalo environment; do not use it to pollute the main Picalo frame.

The `get_menu_items` method returns the menu items for the given plugin. It returns a list of `Utils.MenuItem` objects, which describe menu items, help text, and functions to call when the menu is selected. Picalo automatically creates the wx menu items and events – you just have to create `MenuItem` objects. See the `Detectlets` plugin for an example of this list.

The following is the most basic `__init__.py` file:

```

1 PLUGIN_VERSION = 1.0 def initialize_plugin(mainframe): '''Called to initialize this plugin.'''
   pass
2 def get_menu_items():
3     '''Returns the menu items for this plugin'''
4     return []

```

For a more realistic example, see the `picalo/tools/Detectlets/__init__.py` file.

# Appendix C

## Example Picalo Scripts

This appendix shows several example scripts in Picalo. All of the examples could be done with the GUI menu system, but the scripts show examples of how to do it automatically. These examples should provide users with concrete scripts to follow.

### C.1 Discovery of Outliers

This example uses the z-score calculation to find outliers in a labor rate table. It looks for people getting paid too much for their job type. The input table is a list of payments to workers over time. Each payment line lists the amount, rate, and craft code. The analysis is to first separate the table into separate tables, one for each craft code (it only makes sense to compare the rate of one painter to another painter, not to a welder).

For each craft-code-specific table, it extracts any rates that are at least 7 standard deviations above the average rate for that craft code. The statistical probability of any rate being that high above the average is extremely unlikely, even though z-scores of 7 often occur in the real world (where data is not exactly bell shaped).

```
1 from picalo import *
2 from picalo.lib import stats
3 import time
4
5 # load the data into a table
6 print "Loading..."
7 data = load_csv('LaborRatesAcrossTime.csv', none=0)
8 data.set_type('Rate', float)
9
10 # stratify the table by Craft Code
```

```

11 groups = Grouping.stratify_by_value(data, 'Craft Code')
12
13 # analyze outliers by group
14 for group in groups:
15     outliers = Simple.select_outliers_z(group, 'Rate', 7)
16     if len(outliers) > 0 and outliers[0]['Rate'] != 0.0:
17         print 'Outliers on', group.startkey
18         print '    Count is', len(group)
19         print '    Average is', stats.mean(group['Rate'])
20         print '    Std Dev is', stats.samplestdev(group['Rate'])
21     outliers.view()

```

## C.2 Identifying Phantom Vendors

This example compares addresses in the employee and vendor file (similarnames.py). A common fraud is for employees to set themselves up as phantom vendors and receive payments for bogus transactions. Being the "smart" frausters that they are, most employees use their home address as their vendor address. You can find these phantom vendors by comparing the addresses of employees to addresses of vendors.

Something that complicates the analysis a little bit is slight changes in addresses or city names. For example, "1010 Maple Street" might be the same as "1010 Maple" or "1010 Maple Way" for a certain city. A direct database match would not find these variations, but a fuzzy match would.

The following routine uses Picalo's fuzzymatch routine to find addresses that have a 30 percent match and cities with a 30 percent match. It is effective at finding variations in addresses.

```

1 from picalo import *
2
3 # load the data
4 print 'Loading the data...'
5 employees = load_tsv('employee.txt')
6 vendors = load_tsv('vendor.txt')
7 print len(employees), 'employees,', len(vendors), 'vendors'
8
9 # look for similar addresses
10 print 'Checking each employee against each vendor...'
11 for emp in employees:
12     for vendor in vendors:
13         address_match = Simple.fuzzymatch(emp['address'],
14         vendor['address'])
15         city_match = Simple.fuzzymatch(emp['city'], vendor['city'])
16         if address_match >= .3 and city_match >= .3:
17             print 'Match, address:', address_match, 'city:', city_match
18             print '\t', 'Employee:', emp['f_name'], emp['l_name'],
19             emp['address'], emp['city'], emp['state']
20             print '\t', 'Vendor: ', vendor['vendor_name'],

```



```
21 vendor['address'], vendor['city'], vendor['state']
```

The printout is as follows:

```
1 Loading the data...
2 2010 employees, 125 vendors
3 Checking each employee against each vendor...
4 Match, address: 0.3125 city: 1.0
5     Employee: Hector ORTIZ 260 MIRIA Pacific Palisades CA
6     Vendor:   Home First 1540 Mirias Street Pacific Palisades CA
7 Match, address: 0.5 city: 1.0
8     Employee: Edmund MCKAY 1540 MIRIA Pacific Palisades CA
9     Vendor:   Home First 1540 Mirias Street Pacific Palisades CA
```

**Discussion:** The script first load an employees table and a vendors table. It then goes through two embedded for loops, going through each vendor for each employee. This essentially ensures that each vendor is matched against each employee.

The first match, Hector and Home First, comes up because of the similarity in address. However, this is probably not a phantom vendor. The second one, Edmund and Home First, is essentially at the same address, even though the match is only 50 percent. This could very well be a phantom vendor committing fraud against the company.

An alternative method of writing this code is shown in the following listing (similarnames2.py). Notice how this code embeds the `fuzzymatch` call within the `Simple.custom_match_same()` function. It returns exactly the same results as the first listing. It is actually a bit slower than the first method, but some may argue that its code is more readable.

```
1 from picalo import *
2
3 # load the data
4 print 'Loading the data...'
5 employees = load_tsv('employee.txt')
6 vendors = load_tsv('vendor.txt')
7 print len(employees), 'employees,', len(vendors), 'vendors'
8
9 # match using a custom formula
10 print 'Matching tables based upon address and city.'
11 matches = Simple.custom_match_same(employees, vendors,
12     "Simple.fuzzymatch(rec1['address'], rec2['address']) > .3 and
13     Simple.fuzzymatch(rec1['city'], rec2['city']) > .3")
14 print 'Matches in table 1:'
15 matches[0].view()
16 print 'Matches in table 2:'
17 matches[1].view()
```

### C.3 Finding Unapproved Vendors

Suppose you have a list of approved vendors (`approved_vendors.tsv`) and a file of invoice charges (`charges.tsv`). This example shows how to match the two tables and find charges from vendors who are not approved.

Following are short excerpts from the data files. Note the common Vendor Code field in each file. The routine finds Vendors Codes used in `charges.tsv` that are not in the approved file.

```

1 approved_vendors.tsv (contains 500 fictitious vendors)
2 =====
3 Vendor Code
4 AAC09
5 AAN92
6 ABT73
7 AFN00
8
9 charges.tsv (contains just over 100,0000 fictitious records)
10 =====
11 Invoice #      Date      Purchaser      Vendor Code      Amount
12 2            01/01/2004    Vijay    OJQ26    85351.87
13 27           01/01/2004    Vijay    GYD00    4089.06
14 32           01/01/2004    Adam     JCY21    66005.48
15 65           01/01/2004    Adam     PAY03    94238.55

```

The Picalo script is written as follows (`find_unapproved_vendors.py`):

```

1 from picalo import *
2
3 # load the tables
4 print 'Loading tables...'
5 charges = load_tsv('charges.tsv')
6 approved = load_tsv('approved_vendors.tsv')
7
8 # do the comparison
9 print 'Comparing...'
10 matches = Simple.col_match_diff(charges, approved,
11                                ['Vendor Code', 'Vendor Code'])
12 matches[0].view()

```

**Discussion** The output of the script is shown below. Note that Suzie is the only purchaser who used unapproved vendors: AC1, AC2, and TRS.

```

1 Loading tables...
2 Comparing...
3
4 | Invoice # |      Date      | Purchaser | Vendor Code |      Amount |
5 |-----|-----|-----|-----|-----|
6 | 2535     | 01/02/2004    | Suzie     | AC2         | 87014.77    |
7 | 1065     | 01/21/2004    | Suzie     | AC1         | 97190.02    |
8 | 922      | 01/26/2004    | Suzie     | AC1         | 93405.62    |
9 | 3134     | 03/07/2004    | Suzie     | AC1         | 72200.96    |
10 | 4215     | 03/19/2004    | Suzie     | AC2         | 91043.75    |
11 | 1133     | 06/24/2004    | Suzie     | TRS         | 73207.81    |

```

12	4454	06/24/2004	Suzie	TRS	52568.36
13	1514	08/20/2004	Suzie	AC1	92750.49
14	3687	08/28/2004	Suzie	TRS	90353.67
15	4069	09/05/2004	Suzie	AC1	53865.46
16	534	09/27/2004	Suzie	TRS	52565.91
17	1019	10/16/2004	Suzie	AC1	52743.65
18	1741	12/09/2004	Suzie	TRS	90871.54
19					

## C.4 Stratification and Summarization

Assume you have a list of invoices charges, including the purchaser and vendor. This analysis determines which purchasers and which vendors are receiving the most activity.

Following is a short excerpts from the data file:

```

1 charges.tsv (contains just over 100,0000 fictitious records)
2 =====
3 Invoice #      Date      Purchaser      Vendor Code      Amount
4 2           01/01/2004      Vijay    OJQ26      85351.87
5 27          01/01/2004      Vijay    GYD00      4089.06
6 32          01/01/2004      Adam     JCY21      66005.48
7 65          01/01/2004      Adam     PAY03      94238.55

```

The Picalo script is written as follows (group.py):

```

1 from picalo import *
2
3 # load the data
4 print 'Opening table'
5 charges = load_tsv('charges.tsv')
6
7 # add a numeric column for amount
8 print 'Converting type'
9 charges.set_type('Amount', float)
10
11 # calculate the summary
12 print 'Summarizing by purchaser'
13 summary = Grouping.summarize_by_value(charges, 'Purchaser',
14     sum="sum(group['Amount'])")
15 Simple.sort(summary, True, 'sum')
16 summary.view()
17
18 # calculate the summary again
19 print 'Summarizing by Vendor'
20 summary = Grouping.summarize_by_value(charges, 'Vendor Code',
21     sum="sum(group['Amount'])")
22 Simple.sort(summary, True, 'sum')
23 summary.view()
24
25 # calculate the summary again
26 print 'Summarizing by Purchaser & Vendor'
27 summary = Grouping.summarize_by_value(charges, 'Purchaser', 'Vendor Code',

```

```

28         sum="sum(group['Amount'])"
29 Simple.sort(summary, True, 'sum')
30 summary.view()
31
32 # do a crosstable
33 print 'Crosstab'
34 pivoted = Crosstable.pivot_table(charges, ['Purchaser'], ['Vendor Code'],
35                                   ['Amount'], ["sum(values)"])
36 pivoted.view()

```

**Discussion:** The script is actually four analyses in one script. It runs the following analyses:

1. *Group by Purchaser*: Shows which purchasers spend the most money.
2. *Group by Vendor*: Shows which vendors receive the most purchases.
3. *Group by Purchaser and Vendor*: Matches purchaser to vendor and shows total charges. This is the most powerful analysis, since it shows connections between purchasers and vendors.
4. *Crosstable (a.k.a. Pivot Table)*: An alternative view showing amounts by purchasers and vendors. This analysis is shown only to highlight different ways of doing the same analysis.

The script produces the following output (clipped for space). Note that Vijay spends twice as much as Suzie or Adam. Note also the amount spent on vendor FEK12 and POR77.

```

1 Opening table
2 Converting type
3 Summarizing by purchaser
4 +-----+-----+-----+
5 | StartKey | EndKey | sum |
6 +-----+-----+-----+
7 | Vijay    | Vijay  | 2520610191.66 |
8 | Suzie    | Suzie  | 1268283774.04 |
9 | Adam     | Adam   | 1263838578.11 |
10 +-----+-----+-----+
11 Summarizing by Vendor
12 +-----+-----+-----+
13 | StartKey | EndKey | sum |
14 +-----+-----+-----+
15 | FEK12    | FEK12  | 48322659.34 |
16 | POR77    | POR77  | 22067427.88 |
17 | LWO59    | LWO59  | 12973463.15 |
18 | RYG71    | RYG71  | 12622297.24 |
19 | AXN63    | AXN63  | 12104987.48 |
20 | SOI83    | SOI83  | 12049221.75 |
21 | ALR32    | ALR32  | 12022202.89 |

```

```

22 | OTF93      | OTF93      | 11959404.56 |
23 | UZA27      | UZA27      | 11950721.17 |
24 | BOR16      | BOR16      | 11872846.49 |
25 | JWH59      | JWH59      | 11847171.89 |
26 | ...
27 |-----+-----+-----+
28 | Summarizing by Purchaser & Vendor
29 |-----+-----+-----+
30 |      StartKey      |      EndKey      |      sum      |
31 |-----+-----+-----+
32 | ('Vijay', 'FEK12') | ('Vijay', 'FEK12') | 27540690.32 |
33 | ('Vijay', 'POR77') | ('Vijay', 'POR77') | 12999052.01 |
34 | ('Adam', 'FEK12')  | ('Adam', 'FEK12')  | 10492870.66 |
35 | ('Suzie', 'FEK12') | ('Suzie', 'FEK12') | 10289098.36 |
36 | ('Vijay', 'UZA27') | ('Vijay', 'UZA27') | 6639118.63 |
37 | ('Vijay', 'SOI83') | ('Vijay', 'SOI83') | 6477284.19 |
38 | ('Vijay', 'JWH59') | ('Vijay', 'JWH59') | 6378545.84 |
39 | ('Vijay', 'DBE55') | ('Vijay', 'DBE55') | 6276612.55 |
40 | ('Vijay', 'QIC27') | ('Vijay', 'QIC27') | 6276457.94 |
41 | ...
42 |-----+-----+-----+
43 | Crosstab
44 |-----+-----+-----+
45 |      ('Totals',)      |      ('Adam',)      |      ('Suzie',)      |      ('Vijay',)      |
46 |-----+-----+-----+
47 | ('AAC09',) | ([2353811.8200000003],) | ([2344879.1900000004],) | ([5455730.3399999989],) | ([101
48 | ('AAN92',) | ([2853195.0599999996],) | ([2449171.5499999993],) | ([4138641.6699999985],) | ([944
49 | ('ABT73',) | ([1916310.0600000001],) | ([2610856.3500000001],) | ([5148373.4199999981],) | ([967
50 | ('AC1',) | ([None],) | ([462156.2000000007],) | ([None],) | ([None],) |
51 | ('AC2',) | ([None],) | ([None],) | ([178058.5200000002],) | ([None],) |
52 | ('AFN00',) | ([2596811.4399999999],) | ([2199526.23],) | ([4680914.8499999987],) | ([None],) |
53 | ('AHT14',) | ([2596812.8999999999],) | ([2787890.2199999993],) | ([4744915.6300000018],) | ([101
54 | ('AII47',) | ([2472053.4899999998],) | ([2490243.9300000002],) | ([5553184.0599999987],) | ([101
55 | ('AIZ56',) | ([2743866.1799999992],) | ([3231217.5899999994],) | ([5028244.0799999991],) | ([110
56 | ('AKO54',) | ([3348725.1399999997],) | ([2255068.9599999995],) | ([4947911.6400000006],) | ([None],) |
57 | ...
58 |-----+-----+-----+

```

## C.5 Standardizing the Time Axis

The following script prepares data for trend analysis on the charges table by showing a summary of spending by purchaser for each two-week period during the year. The results could be imported into Excel (after export from Picalo using the `save_tsv()` method) and graphed to show visual trends.

Note that, for this analysis, the *charges.tsv* table is actually three subtables that require separate analysis: purchases for Vijay, purchases for Suzie,

and purchases for Adam. The table is split using the `group_by_key` function, and the subtables are stored in the `groups` list variable. The `for` construct loops through the three tables in the list and iteratively applies the summary analysis.

The script is as follows (`standardize_time_axis.py`):

```

1 from picalo import *
2
3 # load the data
4 print 'Loading'
5 charges = load_tsv('charges.tsv')
6 charges.set_type('Date', DateTime)
7 charges.set_type('Amount', float)
8
9 # split records into a table for each Purchaser
10 print 'Creating groups'
11 groups = Grouping.stratify_by_value(charges, 'Purchaser')
12
13 # summarize into two-week periods
14 print 'Summarizing by purchaser'
15 for group in groups:
16     summary = Grouping.summarize_by_date(group, 'Date', 14,
17         sum="sum(group['Amount'])",
18         count="len(group)")
19     print
20     print 'Purchaser:', group.startkey, sum(summary['sum'])
21     summary.view()

```

The results of the script are as follows (tables have been clipped for space).

The results don't show anything out of the ordinary (at least, to me :).

```

1 Loading
2 Converting types
3 Creating groups
4 Summarizing by purchaser
5
6 Purchaser: Adam 1263838578.11
7 +-----+-----+-----+-----+
8 |           StartKey           |           EndKey           | count |      sum      |
9 +-----+-----+-----+-----+
10 | 2004-01-01 00:00:00.00 | 2004-01-15 00:00:-0.00 | 1076 | 56264155.8 |
11 | 2004-01-15 00:00:-0.00 | 2004-01-29 00:00:-0.00 | 998 | 50826750.89 |
12 | 2004-01-29 00:00:-0.00 | 2004-02-12 00:00:-0.00 | 825 | 40656696.82 |
13 ...
14 +-----+-----+-----+-----+
15
16 Purchaser: Suzie 1268283774.04
17 +-----+-----+-----+-----+
18 |           StartKey           |           EndKey           | count |      sum      |
19 +-----+-----+-----+-----+
20 | 2004-01-01 00:00:00.00 | 2004-01-15 00:00:-0.00 | 1036 | 51525642.64 |
21 | 2004-01-15 00:00:-0.00 | 2004-01-29 00:00:-0.00 | 1062 | 54669823.53 |
22 | 2004-01-29 00:00:-0.00 | 2004-02-12 00:00:-0.00 | 851 | 44050315.38 |
23 ...
24 +-----+-----+-----+-----+
25

```

26 Purchaser: Vijay 2520610191.66

27				
28	StartKey	EndKey	count	sum
29				
30	2004-01-01 00:00:00.00	2004-01-15 00:00:-0.00	2079	105031551.22
31	2004-01-15 00:00:-0.00	2004-01-29 00:00:-0.00	2120	107069633.18
32	2004-01-29 00:00:-0.00	2004-02-12 00:00:-0.00	1654	82064955.03
33	...			
34				

**Discussion:** Databases do not keep records consistently over time. For example, suppose a given table contains a record for each sale transaction. The business might have 1 sale on Monday, 8 sales on Tuesday, 16 sales on Wednesday, 5 sales on Thursday, and so forth. If you want to trend sales for the week, you cannot simply graph the data with an x-point for each of the 30 sales. Instead, you should summarize the total sales on the first day, the total sales for the second day, and graph the totals. This is call *standardizing the time axis*.

In a similar manner, this analysis needs to standardize the time axis to know how many purchases each person made per each unit of time (2 weeks or 14 days in this case). The `summarize_by_date` function is an easy way to standardize your time axis.

The next step in this analysis is to determine whether purchases are increasing (possibly indicating one of several frauds) for any given purchaser. This could be done by graphing the summary tables in a spreadsheet like Excel, or it could be done with one of the trending functions. If you had 1,000 purchasers, graphing each result would be extremely tedious and time consuming. The trending module has functions that calculate a slope (in different ways) for each summary table. Using one of the trending functions, you can look at a single number per purchaser (the slope) and determine whether his or her purchases were increasing.

This is obviously not the only way to discover purchasing fraud, and increasing trends are caused by many more things than fraud. But in any case, the routine helps to focus your attention in datasets that are simply too large to analyze by hand.