Chapter 15. Regular Expressions

Chapter Topics

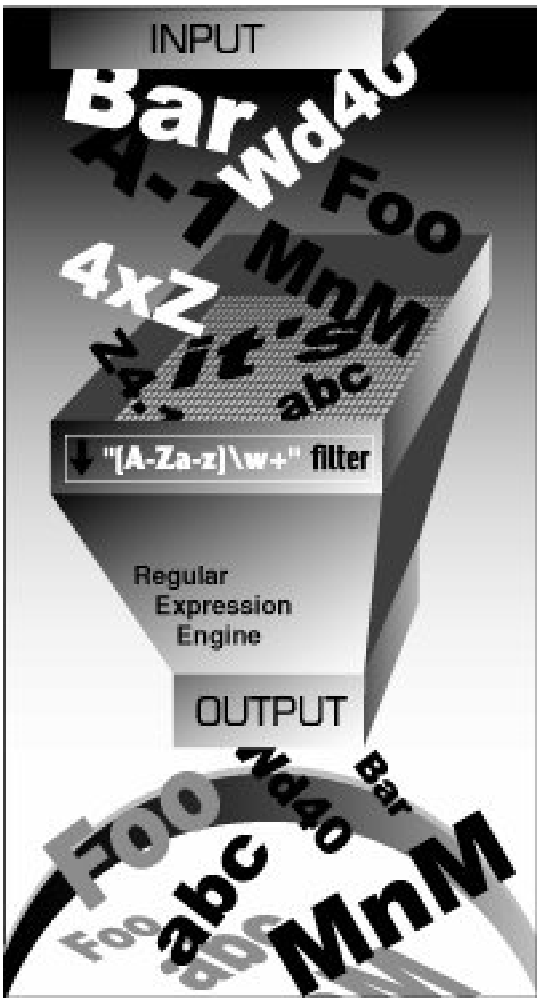
15.1. Introduction/Motivation

Manipulating text/data is a big thing. If you don't believe me, look very carefully at what computers primarily do today. Word processing, "fill-out-form" Web pages, streams of information coming from a database dump, stock quote information, news feedsthe list goes on and on. Because we may not know the exact text or data that we have programmed our machines to process, it becomes advantageous to be able to express this text or data in patterns that a machine can recognize and take action upon.

If I were running an electronic mail (e-mail) archiving company, and you were one of my customers who requested all his or her e-mail sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually. You would be horrified (and infuriated) that someone would be rummaging through your messages, even if his or her eyes were *supposed* to be looking only at time-stamp. Another example request might be to look for a subject line like "ILOVEYOU" indicating a virus-infected message and remove those e-mail messages from your personal archive. So this begs the question of how we can program machines with the ability to look for patterns in text.

Regular expressions (REs) provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality. REs are simply strings that use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern (Figure 15-1). In other words, they enable matching of multiple stringsan RE pattern that matched only one string would be rather boring and ineffective, wouldn't you say?

Figure 15-1. You can use regular expressions, such as the one here, which recognizes valid Python identifiers. "`[A-Za-z]\w+`" means the first character should be alphabetic, i.e., either A-Z or a-z, followed by at least one (+) alphanumeric character (\w). In our filter, notice how many strings go into the filter, but the only ones to come out are the ones we asked for via the RE. One example that did not make it was "4xZ" because it starts with a number.



Python supports REs through the standard library `re` module. In this introductory subsection, we will give you a brief and concise introduction. Due to its brevity, only the most common aspects of REs used in everyday Python programming will be covered. Your experience will, of course, vary. We highly recommend reading any of the official supporting documentation as well as external texts on this interesting subject. You will never look at strings the same way again!

Core Note: Searching versus matching

Throughout this chapter, you will find references to searching and matching. When we are strictly discussing regular expressions with respect to patterns in strings, we will say "matching," referring to the term pattern-matching. In Python terminology, there are two main ways to accomplish pattern-matching: searching, i.e., looking for a pattern match in any part of a string, and matching, i.e., attempting to match a pattern to an entire string (starting from the beginning). Searches are accomplished using the `search()` *function or method, and matching is done with the* `match()` *function or method. In summary, we keep the term "matching" universal when referencing patterns, and we differentiate between "searching" and "matching" in terms of how Python accomplishes pattern-matching.*

15.1.1. Your First Regular Expression

As we mentioned above, REs are strings containing text and special characters that describe a pattern with which to recognize multiple strings. We also briefly discussed a regular expression *alphabet* and for general text, the alphabet used for regular expressions is the set of all uppercase and lowercase letters plus numeric digits. Specialized alphabets are also possible, for instance, one consisting of only the characters "0" and "1". The set of all strings over this alphabet describes all binary strings, i.e., "0," "1," "00," "01," "10," "11," "100," etc.

Let us look at the most basic of regular expressions now to show you that although REs are sometimes considered an "advanced topic," they can also be rather simplistic. Using the standard alphabet for general text, we present some simple REs and the strings that their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern that matches only one string, the string defined by the regular expression. We now present the REs followed by the strings that match them:

RE Pattern	String(s) Matched
`foo`	`foo`
`Python`	`Python`
`abc123`	`abc123`

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string that matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow an RE to match a set of strings rather than a single one.

PREV NEXT

15.2. Special Symbols and Characters

We will now introduce the most popular of the *metacharacters*, special characters and symbols, which give regular expressions their power and flexibility. You will find the most common of these symbols and characters in Table 15.1.

Table 15.1. Common Regular Expression Symbols and Special Characters

Notation	***Description***	***Example RE***
Symbols		
`literal`	Match literal string value `literal`	`foo`
`re1|re2`	Match regular expressions `re1` or `re2`	`foo|bar`
`.`	Match *any character* (except NEWLINE)	`b.b`
`^`	Match *start of string*	`^Dear`
`$`	Match *end of string*	`/bin/*sh$`
`*`	Match *0 or more* occurrences of preceding RE	`[A-Za-z0-9]*`
`+`	Match *1 or more* occurrences of preceding RE	`[a-z]+\.com`
`?`	Match *0 or 1* occurrence(s) of preceding RE	`goo?`
`{N}`	Match `N` occurrences of preceding RE	`[0-9]{3}`
`{M,N}`	Match from `M` to `N` occurrences of preceding RE	`[0-9]{5,9}`
`[...]`	Match any single character from *character* class	`[aeiou]`
`[..x-y..]`	Match any single character in the *range from* `x` to `y`	`[0-9]`, `[A-Za-z]`
`[^...]`	*Do not match* any character from character class, including any ranges, if present	`[^aeiou]`, `[^A-Za-z0-9_]`
`(*|+|?| {})?`	Apply "non-greedy" versions of above occurrence/repetition symbols (`*`, `+`, `?`, `{}`)	`.*?[a-z]`
`(...)`	Match enclosed RE and save as *subgroup*	`([0-9]{3})?`, `f(oo|u)bar`
Special Characters		
`\d`	Match any decimal *digit*, same as `[0-9]`(`\D` is inverse of `\d`: do not match any numeric digit)	`data\d+.txt`
`\w`	Match any *alphanumeric* character, same as `[A-Za-z0-9_]` (`\W` is inverse of `\w`)	`[A-Za-z_]\w+`

`\s`	Match *any whitespace* character, same as `[ \n\t\r\v \f]` (`\S` is inverse of `\s`)	`of\sthe`
`\b`	Match any *word boundary* (`\B` is inverse of `\b`)	`\bThe\b`
`\`*`nn`*	Match saved *subgroup* *`nn`* (see `(...)` above)	`price: \16`
`\`*`c`*	Match any *special character* *`c`* verbatim (i.e., with out its special meaning, literal)	`\., \\, \*`
`\A (\Z)`	Match *start (end) of string* (also see `^` and `$` above)	`\ADear`

15.2.1. Matching More Than One RE Pattern with Alternation (|)

The pipe symbol (`|`), a vertical bar on your keyboard, indicates an *alternation* operation, meaning that it is used to choose from one of the different regular expressions, which are separated by the pipe symbol. For example, below are some patterns that employ alternation, along with the strings they match:

RE Pattern	Strings Matched
`at|home`	`at, home`
`r2d2|c3po`	`r2d2, c3po`
`bat|bet|bit`	`bat, bet, bit`

With this one symbol, we have just increased the flexibility of our regular expressions, enabling the matching of more than just one string. Alternation is also sometimes called union or logical OR.

15.2.2. Matching Any Single Character (.)

The dot or period (`.`) symbol matches any single character except for NEWLINE (Python REs have a compilation flag [`S` or `DOTALL`], which can override this to include NEWLINEs.). Whether letter, number, whitespace not including "`\n`," printable, non-printable, or a symbol, the dot can match them all.

RE Pattern	Strings Matched
`f.o`	Any character between "f" and "o", e.g., `fao`, `f9o`, `f#o`, `etc.`
`..`	Any pair of characters
`.end`	Any character before the string `end`

Q: What if I want to match the dot or period character?

A: In order to specify a dot character explicitly, you must escape its functionality with a backslash, as in

"\.".

15.2.3. Matching from the Beginning or End of Strings or Word Boundaries (^/$ /\b / \B)

There are also symbols and related special characters to specify searching for patterns at the beginning and ending of strings. To match a pattern starting from the beginning, you must use the carat symbol (^) or the special character \A (backslash-capital "A"). The latter is primarily for keyboards that do not have the carat symbol, i.e., international. Similarly, the dollar sign ($) or \Z will match a pattern from the end of a string.

Patterns that use these symbols differ from most of the others we describe in this chapter since they dictate location or position. In the Core Note above, we noted that a distinction is made between "matching," attempting matches of entire strings starting at the beginning, and "searching," attempting matches from anywhere within a string. With that said, here are some examples of "edge-bound" RE search patterns:

RE Pattern	Strings Matched
`^From`	Any string that starts with `From`
`/bin/tcsh$`	Any string that ends with `/bin/tcsh`
`^Subject: hi$`	Any string consisting solely of the string `Subject: hi`

Again, if you want to match either (or both) of these characters verbatim, you must use an escaping backslash. For example, if you wanted to match any string that ended with a dollar sign, one possible RE solution would be the pattern ".*\$$".

The \b and \B special characters pertain to word boundary matches. The difference between them is that \b will match a pattern to a word boundary, meaning that a pattern must be at the beginning of a word, whether there are any characters in front of it (word in the middle of a string) or not (word at the beginning of a line). And likewise, \B will match a pattern only if it appears starting in the middle of a word (i.e., not at a word boundary). Here are some examples:

RE Pattern	Strings Matched
`the`	Any string containing `the`
`\bthe`	Any word that starts with `the`
`\bthe\b`	Matches only the word `the`
`\Bthe`	Any string that contains but does not begin with `the`

15.2.4. Creating Character Classes ([ ])

While the dot is good for allowing matches of any symbols, there may be occasions where there are specific characters you want to match. For this reason, the bracket symbols (`[ ]`) were invented. The regular expression will match any of the enclosed characters. Here are some examples:

RE Pattern	Strings Matched
`b[aeiu]t`	`bat`, `bet`, `bit`, `but`
`[cr][23][dp][o2]`	A string of 4 characters: first is "r" or "c," then "2" or "3," followed by "d" or "p," and finally, either "o" or "2," e.g., `c2do`, `r3p2`, `r2d2`, `c3po`, etc.

One side note regarding the RE "`[cr][23][dp][o2]`"a more restrictive version of this RE would be required to allow only "r2d2" or "c3po" as valid strings. Because brackets merely imply "logical OR" functionality, it is not possible to use brackets to enforce such a requirement. The only solution is to use the pipe, as in "`r2d2|c3po`".

For single-character REs, though, the pipe and brackets are equivalent. For example, let's start with the regular expression "ab," which matches only the string with an "a" followed by a "b". If we wanted either a one-letter string, i.e., either "a" or a "b," we could use the RE "`[ab]`." Because "a" and "b" are individual strings, we can also choose the RE "`a|b`". However, if we wanted to match the string with the pattern "ab" followed by "cd," we cannot use the brackets because they work only for single characters. In this case, the only solution is "`ab|cd`," similar to the "`r2d2/c3po`" problem just mentioned.

15.2.5. Denoting Ranges (`-`) and Negation (`^`)

In addition to single characters, the brackets also support ranges of characters. A hyphen between a pair of symbols enclosed in brackets is used to indicate a range of characters, e.g., A-Z, a-z, or 0-9 for uppercase letters, lowercase letters, and numeric digits, respectively. This is a lexicographic range, so you are not restricted to using just alphanumeric characters. Additionally, if a caret (`^`) is the first character immediately inside the open left bracket, this symbolizes a directive *not* to match any of the characters in the given character set.

RE Pattern	Strings Matched
`z.[0-9]`	"z" followed by any character then followed by a single digit
`[r-u][env-y]`	"r" "s," "t" or "u" followed by "e," "n," "v," "w," "x," or "y"
`[us]`	followed by "u" or "s"
`[^aeiou]`	A non-vowel character (Exercise: Why do we say "non-vowels" rather than "consonants"?)
`[^\t\n]`	Not a TAB or NEWLINE
`["-a]`	In an ASCII system, all characters that fall between '"' and "a," i.e., between ordinals 34 and 97

15.2.6. Multiple Occurrence/Repetition Using Closure Operators (`*`, `+`, `?`, `{ }`)

We will now introduce the most common RE notations, namely, the special symbols `*`, `+`, and `?`, all of which can be used to match single, multiple, or no occurrences of string patterns. The asterisk or star operator (`*`) will match zero or more occurrences of the RE immediately to its left (in language and compiler theory, this operation is known as the *Kleene Closure*). The plus operator (`+`) will match one or more occurrences of an RE (known as *Positive Closure*), and the question mark operator (`?`) will match exactly 0 or 1 occurrences of an RE.

There are also brace operators (`{ }`) with either a single value or a comma-separated pair of values. These indicate a match of exactly *N* occurrences (for `{N}`) or a range of occurrences, i.e., `{M,N}` will match from *M* to *N* occurrences. These symbols may also be escaped with the backslash, i.e., "`\*`" matches the asterisk, etc.

In the table above, we notice the question mark is used more than once (overloaded), meaning either matching 0 or 1 occurrences, or its other meaning: if it follows any matching using the close operators, it will direct the regular expression engine to match as few repetitions as possible.

What does that last part mean, "as few ... as possible?" When pattern-matching is employed using the grouping operators, the regular expression engine will try to "absorb" as many characters as possible which match the pattern. This is known as being *greedy*. The question mark tells the engine to lay off and if possible, take as few characters as possible in the current match, leaving the rest to match as many of succeeding characters of the next pattern (if applicable). We will show you a great example where non-greediness is required toward the end of the chapter. For now, let us continue to look at the closure operators:

RE Pattern	**Strings Matched**
`[dn]ot?`	"d" or "n," followed by an "o" and, at most, one "t" after that, i.e., `do`, `no`, `dot`, `not`
`0?[1-9]`	Any numeric digit, possibly prepended with a "0," e.g., the set of numeric representations of the months January to September, whether single- or double-digits
`[0-9]{15,16}`	Fifteen or sixteen digits, e.g., credit card numbers
`</?[^>]+>`	Strings that match all valid (and invalid) HTML tags
`[KQRBNP][a-h][1-8]-[a-h][1-8]`	Legal chess move in "long algebraic" notation (move only, no capture, check, etc.), i.e., strings which start with any of "K," "Q," "R," "B," "N," or "P" followed by a hyphenated-pair of chess board grid locations from "a1" to "h8" (and everything in between), with the first coordinate indicating the former position and the second being the new position.

15.2.7. Special Characters Representing Character Sets

We also mentioned that there are special characters that may represent character sets. Rather than using a range of "0-9," you may simply use "`\d`" to indicate the match of any decimal digit. Another special character "`\w`" can be used to denote the entire alphanumeric character class, serving as a shortcut for "`A-Za-z0-9_`", and "`\s`" for whitespace characters. Uppercase versions of these strings symbolize *non*-matches, i.e., "`\D`" matches any non-decimal digit (same as "`[^0-9]`"), etc.

Using these shortcuts, we will present a few more complex examples:

RE Pattern	Strings Matched
`\w+-\d+`	Alphanumeric string and number separated by a hyphen
`[A-Za-z]\w*`	Alphabetic first character, additional characters (if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers [see exercises])
`\d{3}-\d{3}-\d{4}`	(American) telephone numbers with an area code prefix, as in 800-555-1212
`\w+@\w+\.com`	Simple e-mail addresses of the form *`XXX@YYY.com`*

15.2.8. Designating Groups with Parentheses (`( )`)

Now, perhaps we have achieved the goal of matching a string and discarding non-matches, but in some cases, we may also be more interested in the data that we did match. Not only do we want to know whether the entire string matched our criteria, but also whether we can extract any specific strings or substrings that were part of a successful match. The answer is yes. To accomplish this, surround any RE with a pair of parentheses.

A pair of parentheses (`( )`) can accomplish either (or both) of the below when used with regular expressions:

- Grouping regular expressions
- Matching subgroups

One good example for wanting to group regular expressions is when you have two different REs with which you want to compare a string. Another reason is to group an RE in order to use a repetition operator on the entire RE (as opposed to an individual character or character class).

One side effect of using parentheses is that the substring that matched the pattern is saved for future use. These subgroups can be recalled for the same match or search, or extracted for post-processing. You will see some examples of pulling out subgroups at the end of Section 15.3.9.

Why are matches of subgroups important? The main reason is that there are times where you want to extract the patterns you match, in addition to making a match. For example, what if we decided to match the pattern "`\w+-\d+`" but wanted save the alphabetic first part and the numeric second part individually? This may be desired because with any successful match, we may want to see just what those strings were that matched our RE patterns.

If we add parentheses to both subpatterns, i.e., "`(\w+)-(\d+)`," then we can access each of the matched subgroups individually. Subgrouping is preferred because the alternative is to write code to determine we have a match, then execute another separate routine (which we also had to create) to parse the entire match just to extract both parts. Why not let Python do it, since it is a supported feature of the `re` module, instead of reinventing the wheel?

RE Pattern	Strings Matched

`\d+(\.\d*)?`	Strings representing simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc.
`(Mr?s?\. )?[A-Z][a-z]* [ A-Za-z-]+`	First name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters

PREV NEXT

PREV NEXT

15.3. REs and Python

Now that we know all about regular expressions, we can examine how Python currently supports regular expressions through the `re` module. The `re` module was introduced to Python in version 1.5. If you are using an older version of Python, you will have to use the now-obsolete `regex` and `regsub` modulesthese older modules are more Emacs-flavored, are not as full-featured, and are in many ways incompatible with the current `re` module. Both modules were removed from Python in 2.5, and import either of the modules from 2.5 and above triggers `Import Error` exception.

2.5

However, regular expressions are still regular expressions, so most of the basic concepts from this section can be used with the old `regex` and `regsub` software. In contrast, the new `re` module supports the more powerful and regular Perl-style (Perl5) REs, allows multiple threads to share the same compiled RE objects, and supports named subgroups. In addition, there is a transition module called `reconvert` to help developers move from `regex/regsub` to `re`. However, be aware that although there are different flavors of regular expressions, we will primarily focus on the current incarnation for Python.

The `re` engine was rewritten in 1.6 for performance enhancements as well as adding Unicode support. The interface was not changed, hence the reason the module name was left alone. The new `re` engineknown internally as `sre` thus replaces the existing 1.5 engineinternally called `pcre`.

15.3.1. `re` Module: Core Functions and Methods

The chart in Table 15.2 lists the more popular functions and methods from the `re` module. Many of these functions are also available as methods of compiled regular expression objects "regex objects" and RE "match objects." In this subsection, we will look at the two main functions/methods, `match()` and `search()`, as well as the `compile()` function. We will introduce several more in the next section, but for more information on all these and the others that we do not cover, we refer you to the Python documentation.

Table 15.2. Common Regular Expression Functions and Methods

Function/Method	***Description***
`re` ***Module Function Only***	
`compile(`*`pattern, flags`*`=0)`	Compile RE *`pattern`* with any optional *`flags`* and return a regex object
`re` ***Module Functions and regex Object Methods***	

`match(`*`pattern, string, flags`*`=0)`	Attempt to match RE *`pattern`* to *`string`* with optional *`flags`*; return match object on success, `None` on failure
`search(`*`pattern, string, flags`*`=0)`	Search for first occurrence of RE *`pattern`* within *`string`* with optional *`flags`*; return match object on success, `None` on failure
`findall(`*`pattern, string[,flags]`*`)`[a]	Look for all (non-overlapping) occurrences of *`pattern`* in *`string`*; return a list of matches
`finditer(`*`pattern, string[, flags]`*`)`[b]	Same as `findall()` except returns an iterator instead of a list; for each match, the iterator returns a match object
`split(`*`pattern, string, max`*`=0)`	Split *`string`* into a list according to RE *`pattern`* delimiter and return list of successful matches, splitting at most *`max`* times (split all occurrences is the default)
`sub(`*`pattern, repl, string, max`*`=0)`	Replace all occurrences of the RE *`pattern`* in *`string`* with *`repl`*, substituting all occurrences unless *`max`* provided (also see `subn()` which, in addition, returns the number of substitutions made)
Match Object Methods	
`group(`*`num`*`=0)`	Return entire match (or specific subgroup *`num`*)
`groups()`	Return all matching subgroups in a tuple (empty if there weren't any)

[a] New in Python 1.5.2; *`flags`* parameter added in 2.4.

[b] New in Python 2.2; *`flags`* parameter added in 2.4.

Core Note: RE compilation (to compile or not to compile?)

*In Chapter 14, we described how Python code is eventually compiled into bytecode, which is then executed by the interpreter. In particular, we mentioned that calling `eval()` or **`exec`** with a code object rather than a string provides a significant performance improvement due to the fact that the compilation process does not have to be performed. In other words, using precompiled code objects is faster than using strings because the interpreter will have to compile it into a code object (anyway) before execution.*

The same concept applies to REsregular expression patterns must be compiled into regex objects before any pattern matching can occur. For REs, which are compared many times during the course of execution, we highly recommend using precompilation first because, again, REs have to be compiled anyway, so doing it ahead of time is prudent for performance reasons. `re.compile()` provides this functionality.

The module functions do cache the compiled objects, though, so it's not as if every `search()` and `match()` with the same RE pattern requires compilation. Still, you save the cache lookups and do not have to *make function calls with the same string over and over. In Python 1.5.2, this cache held up to 20 compiled RE objects, but in 1.6, due to the additional overhead of Unicode awareness, the compilation engine is a bit slower, so the cache has been extended to 100 compiled regex objects.*

15.3.2. Compiling REs with `compile()`

Almost all of the `re` module functions we will be describing shortly are available as methods for regex objects. Remember, even with our recommendation, precompilation is not required. If you compile, you will use methods; if you don't, you will just use functions. The good news is that either way, the names are the same whether a function or a method. (This is the reason why there are module functions and methods that are identical, e.g., `search()`, `match()`, etc., in case you were wondering.) Since it saves one small step for most of our examples, we will use strings instead. We will throw in a few with compilation, though, just so you know how it is done.

Optional flags may be given as arguments for specialized compilation. These flags allow for case-insensitive matching, using system locale settings for matching alphanumeric characters, etc. Please refer to the documentation for more details. These flags, some of which have been briefly mentioned (i.e., `DOTALL`, `LOCALE`), may also be given to the module versions of `match()` and `search()` for a specific pattern match attemptthese flags are mostly for compilation reasons, hence the reason why they can be passed to the module versions of `match()` and `search()`, which do compile an RE pattern once. If you want to use these flags with the methods, they must already be integrated into the compiled regex objects.

In addition to the methods below, regex objects also have some data attributes, two of which include any compilation flags given as well as the regular expression pattern compiled.

15.3.3. Match Objects and the `group()` and `groups ()` Methods

There is another object type in addition to the regex object when dealing with regular expressions, the *match object*. These are the objects returned on successful calls to `match()` or `search()`. Match objects have two primary methods, `group()` and `groups()`.

`group()` will either return the entire match, or a specific subgroup, if requested. `groups()` will simply return a tuple consisting of only/all the subgroups. If there are no subgroups requested, then `groups()` returns an empty tuple while `group()` still returns the entire match.

Python REs also allow for named matches, which are beyond the scope of this introductory section on REs. We refer you to the complete `re` module documentation regarding all the more advanced details we have omitted here.

15.3.4. Matching Strings with `match()`

`match()` is the first `re` module function and RE object (regex object) method we will look at. The `match()` function attempts to match the pattern to the string, starting at the beginning. If the match is successful, a match object is returned, but on failure, `None` is returned. The `group()` method of a match object can be used to show the successful match. Here is an example of how to use `match()` [and `group ()`]:

```
>>> m = re.match('foo', 'foo')    # pattern matches string
>>> if m is not None:         # show match if successful
...      m.group()
...
'foo'
```

The pattern "foo" matches exactly the string "foo." We can also confirm that `m` is an example of a match object from within the interactive interpreter:

```
>>> m                  # confirm match object returned
<re.MatchObject instance at 80ebf48>
```

Here is an example of a failed match where `None` is returned:

```
>>> m = re.match('foo', 'bar')# pattern does not match string
>>> if m is not None: m.group()# (1-line version of if
clause)
...
>>>
```

The match above fails, thus `None` is assigned to `m`, and no action is taken due to the way we constructed our **`if`** statement. For the remaining examples, we will try to leave out the **`if`** check for brevity, if possible, but in practice it is a good idea to have it there to prevent `AttributeError` exceptions (`None` is returned on failures, which does not have a `group()` attribute [method].)

A match will still succeed even if the string is longer than the pattern as long as the pattern matches from the beginning of the string. For example, the pattern "foo" will find a match in the string "food on the table" because it matches the pattern from the beginning:

```
>>> m = re.match('foo', 'food on the table') # match succeeds
>>> m.group()
'foo'
```

As you can see, although the string is longer than the pattern, a successful match was made from the beginning of the string. The substring "foo" represents the match, which was extracted from the larger string.

We can even sometimes bypass saving the result altogether, taking advantage of Python's object-oriented nature:

```
>>> re.match('foo', 'food on the table').group()
'foo'
```

Note from a few paragraphs above that an `AttributeError` will be generated on a non-match.

15.3.5. Looking for a Pattern within a String with `search()` (Searching versus Matching)

The chances are greater that the pattern you seek is somewhere in the middle of a string, rather than at the beginning. This is where `search()` comes in handy. It works exactly in the same way as match except that it searches for the first occurrence of the given RE pattern anywhere with its string argument. Again, a match object is returned on success and `None` otherwise.

We will now illustrate the difference between `match()` and `search()`. Let us try a longer string match attempt. This time, we will try to match our string "foo" to "seafood":

```
>>> m = re.match('foo', 'seafood')     # no match
>>> if m is not None: m.group()
...
>>>
```

As you can see, there is no match here. `match()` attempts to match the pattern to the string from the beginning, i.e., the "f" in the pattern is matched against the "s" in the string, which fails immediately. However, the string "foo" *does* appear (elsewhere) in "seafood," so how do we get Python to say "yes"? The answer is by using the `search()` function. Rather than attempting a *match*, `search()` looks for the first occurrence of the pattern within the string. `search()` searches strictly from left to right.

```
>>> m = re.search('foo', 'seafood')   # use search() instead
>>> if m is not None: m.group()
...
'foo'                   # search succeeds where match failed
>>>
```

We will be using the `match()` and `search()` regex object methods and the `group()` and `groups()` match object methods for the remainder of this subsection, exhibiting a broad range of examples of how to use regular expressions with Python. We will be using almost all of the special characters and symbols that are part of the regular expression syntax.

15.3.6. Matching More than One String (`|`)

In Section 15.2, we used the pipe in the RE "`bat|bet|bit`." Here is how we would use that RE with Python:

```
>>> bt = 'bat|bet|bit'       # RE pattern: bat, bet, bit
>>> m = re.match(bt, 'bat')       # 'bat' is a match
>>> if m is not None: m.group()
...
'bat'
>>> m = re.match(bt, 'blt')       # no match for 'blt'
>>> if m is not None: m.group()
...
>>> m = re.match(bt, 'He bit me!') # does not match string
>>> if m is not None: m.group()
...
>>> m = re.search(bt, 'He bit me!') # found 'bit' via search
>>> if m is not None: m.group()
...
'bit'
```

15.3.7. Matching Any Single Character (`.`)

In the examples below, we show that a dot cannot match a NEWLINE or a non-character, i.e., the empty string:

```
>>> anyend = '.end'
>>> m = re.match(anyend, 'bend')     # dot matches 'b'
>>> if m is not None: m.group()
...
'bend'
>>> m = re.match(anyend, 'end')      # no char to match
>>> if m is not None: m.group()
...
>>> m = re.match(anyend, '\nend')    # any char except \n
>>> if m is not None: m.group()
...
>>> m = re.search('.end', 'The end.') # matches ' ' in search
>>> if m is not None: m.group()
...
' end'
```

The following is an example of searching for a real dot (decimal point) in a regular expression where we escape its functionality with a backslash:

```
 >>> patt314 = '3.14'         # RE dot
 >>> pi_patt = '3\.14'        # literal dot (dec. point)
>>> m = re.match(pi_patt, '3.14') # exact match
>>> if m is not None: m.group()
...
'3.14'
>>> m = re.match(patt314, '3014') # dot matches '0'
>>> if m is not None: m.group()
...
'3014'
>>> m = re.match(patt314, '3.14') # dot matches '.'
>>> if m is not None: m.group()
...
'3.14'
```

15.3.8. Creating Character Classes (`[ ]`)

Earlier, we had a long discussion about "`[cr][23][dp][o2]`" and how it differs from "`r2d2|c3po`." With the examples below, we will show that "`r2d2|c3po`" is more restrictive than "`[cr][23][dp][o2]`":

```
>>> m = re.match('[cr][23][dp][o2]', 'c3po') # matches 'c3po'
>>> if m is not None: m.group()
...
'c3po'
>>> m = re.match('[cr][23][dp][o2]', 'c2do') # matches 'c2do'
>>> if m is not None: m.group()
...
'c2do'
```

```
>>> m = re.match('r2d2|c3po', 'c2do') # does not match 'c2do'
>>> if m is not None: m.group()
...
>>> m = re.match('r2d2|c3po', 'r2d2') # matches 'r2d2'
>>> if m is not None: m.group()
...
'r2d2'
```

15.3.9. Repetition, Special Characters, and Grouping

The most common aspects of REs involve the use of special characters, multiple occurrences of RE patterns, and using parentheses to group and extract submatch patterns. One particular RE we looked at related to simple e-mail addresses ("`\w+@\w+\.com`"). Perhaps we want to match more e-mail addresses than this RE allows. In order to support an additional hostname in front of the domain, i.e., "`www.xxx.com`" as opposed to accepting only "`xxx.com`" as the entire domain, we have to modify our existing RE. To indicate that the hostname is optional, we create a pattern that matches the hostname (followed by a dot), use the `?` operator indicating zero or one copy of this pattern, and insert the optional RE into our previous RE as follows: "`\w+@(\w+\.)?\w+\.com.`" As you can see from the examples below, either one or two names are now accepted in front of the "`.com`":

```
>>> patt = '\w+@(\w+\.)?\w+\.com'
>>> re.match(patt, 'nobody@xxx.com').group()
'nobody@xxx.com'
>>> re.match(patt, 'nobody@www.xxx.com').group()
'nobody@www.xxx.com'
```

Furthermore, we can even extend our example to allow any number of intermediate subdomain names with the pattern below. Take special note of our slight change from using `?` to `*.`: "`\w+@(\w+\.)*\w+\.com`":

```
>>> patt = '\w+@(\w+\.)*\w+\.com'
>>> re.match(patt, 'nobody@www.xxx.yyy.zzz.com').group()
'nobody@www.xxx.yyy.zzz.com'
```

However, we must add the disclaimer that using solely alphanumeric characters does not match all the possible characters that may make up e-mail addresses. The above RE patterns would not match a domain such as "`xxx-yyy.com`" or other domains with "`\W`" characters.

Earlier, we discussed the merits of using parentheses to match and save subgroups for further processing rather than coding a separate routine to manually parse a string after an RE match had been determined. In particular, we discussed a simple RE pattern of an alphanumeric string and a number separated by a hyphen, "`\w+-\d+,`" and how adding subgrouping to form a new RE, "`(\w+)-(\d+),`" would do the job. Here is how the original RE works:

```
>>> m = re.match('\w\w\w-\d\d\d', 'abc-123')
>>> if m is not None: m.group()
...
'abc-123'

>>> m = re.match('\w\w\w-\d\d\d', 'abc-xyz')
>>> if m is not None: m.group()
```

```
...
>>>
```

In the above code, we created an RE to recognize three alphanumeric characters followed by three digits. Testing this RE on "`abc-123`," we obtained positive results while "`abc-xyz`" fails. We will now modify our RE as discussed before to be able to extract the alphanumeric string and number. Note how we can now use the `group()` method to access individual subgroups or the `groups()` method to obtain a tuple of all the subgroups matched:

```
>>> m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')
>>> m.group()                          # entire match
'abc-123'
>>> m.group(1)                         # subgroup 1
'abc'
>>> m.group(2)                         # subgroup 2
'123'
>>> m.groups()                         # all subgroups
('abc', '123')
```

As you can see, `group()` is used in the normal way to show the entire match, but can also be used to grab individual subgroup matches. We can also use the `groups()` method to obtain a tuple of all the substring matches.

Here is a simpler example showing different group permutations, which will hopefully make things even more clear:

```
>>> m = re.match('ab', 'ab')           # no subgroups
>>> m.group()                          # entire match
'ab'
>>> m.groups()                         # all subgroups
()
>>>
>>> m = re.match('(ab)', 'ab')         # one subgroup
>>> m.group()                          # entire match
'ab'
>>> m.group(1)                         # subgroup 1
'ab'
>>> m.groups()                         # all subgroups
('ab',)
>>>
>>> m = re.match('(a)(b)', 'ab')            # two subgroups
>>> m.group()                          # entire match
'ab'
>>> m.group(1)                         # subgroup 1
'a'
>>> m.group(2)                         # subgroup 2
'b'
>>> m.groups()                         # all subgroups

('a', 'b')
>>>
>>> m = re.match('(a(b))', 'ab')             # two subgroups
>>> m.group()                          # entire match
'ab'
>>> m.group(1)                         # subgroup 1
```

```
'ab'
>>> m.group(2)                    # subgroup 2
'b'
>>> m.groups()                    # all subgroups
('ab', 'b')
```

15.3.10. Matching from the Beginning and End of Strings and on Word Boundaries

The following examples highlight the positional RE operators. These apply more for searching than matching because `match()` always starts at the beginning of a string.

```
>>> m = re.search('^The', 'The end.')      # match
>>> if m is not None: m.group()
...
'The'
>>> m = re.search('^The', 'end. The')         # not at beginning
>>> if m is not None: m.group()
...
>>> m = re.search(r'\bthe', 'bite the dog') # at a boundary
>>> if m is not None: m.group()
...
'the'
>>> m = re.search(r'\bthe', 'bitethe dog')  # no boundary
>>> if m is not None: m.group()
...
>>> m = re.search(r'\Bthe', 'bitethe dog')  # no boundary

>>> if m is not None: m.group()
...
'the'
```

You will notice the appearance of raw strings here. You may want to take a look at the Core Note toward the end of the chapter for clarification on why they are here. In general, it is a good idea to use raw strings with regular expressions.

There are four other `re` module functions and regex object methods we think you should be aware of: `findall()`, `sub()`, `subn()`, and `split()`.

15.3.11. Finding Every Occurrence with `findall()`

`findall()` is new to Python as of version 1.5.2. It looks for all non-overlapping occurrences of an RE pattern in a string. It is similar to `search()` in that it performs a string search, but it differs from `match()` and `search()` in that `findall()` always returns a list. The list will be empty if no occurrences are found but if successful, the list will consist of all matches found (grouped in left-to-right order of occurrence).

```
>>> re.findall('car', 'car')
['car']
>>> re.findall('car', 'scary')
['car']
>>> re.findall('car', 'carry the barcardi to the car')
['car', 'car', 'car']
```

Subgroup searches result in a more complex list returned, and that makes sense, because subgroups are a mechanism that allow you to extract specific patterns from within your single regular expression, such as matching an area code that is part of a complete telephone number, or a login name that is part of an entire e-mail address.

For a single successful match, each subgroup match is a single element of the resulting list returned by `findall()`; for multiple successful matches, each subgroup match is a single element in a tuple, and such tuples (one for each successful match) are the elements of the resulting list. This part may sound confusing at first, but if you try different examples, it will help clarify things.

15.3.12. Searching and Replacing with `sub()` [and `subn()`]

There are two functions/methods for search-and-replace functionality: `sub()` and `subn()`. They are almost identical and replace all matched occurrences of the RE pattern in a string with some sort of replacement. The replacement is usually a string, but it can also be a function that returns a replacement string. `subn()` is exactly the same as `sub()`, but it also returns the total number of substitutions madeboth the newly substituted string and the substitution count are returned as a 2-tuple.

```
>>> re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
'attn: Mr. Smith\012\012Dear Mr. Smith,\012'
>>>
>>> re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')

('attn: Mr. Smith\012\012Dear Mr. Smith,\012', 2)
>>>
>>> print re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
attn: Mr. Smith

Dear Mr. Smith,

>>> re.sub('[ae]', 'X', 'abcdef')
'XbcdXf'
>>> re.subn('[ae]', 'X', 'abcdef')
('XbcdXf', 2)
```

15.3.13. Splitting (on Delimiting Pattern) with `split()`

The `re` module and RE object method `split()` work similarly to its string counterpart, but rather than splitting on a fixed string, they split a string based on an RE pattern, adding some significant power to string splitting capabilities. If you do not want the string split for every occurrence of the pattern, you can specify the maximum number of splits by setting a value (other than zero) to the `max` argument.

If the delimiter given is not a regular expression that uses special symbols to match multiple patterns, then `re.split()` works in exactly the same manner as `string.split()`, as illustrated in the example below (which splits on a single colon):

```
>>> re.split(':', 'str1:str2:str3')
['str1', 'str2', 'str3']
```

But with regular expressions involved, we have an even more powerful tool. Take, for example, the output from the Unix `who` command, which lists all the users logged into a system:

```
% who
wesc      console      Jun 20 20:33
wesc      pts/9        Jun 22 01:38    (192.168.0.6)
wesc      pts/1        Jun 20 20:33    (:0.0)
wesc      pts/2        Jun 20 20:33    (:0.0)
wesc      pts/4        Jun 20 20:33    (:0.0)
wesc      pts/3        Jun 20 20:33    (:0.0)
wesc      pts/5        Jun 20 20:33    (:0.0)
wesc      pts/6        Jun 20 20:33    (:0.0)
wesc      pts/7        Jun 20 20:33    (:0.0)
wesc      pts/8        Jun 20 20:33    (:0.0)
```

Perhaps we want to save some user login information such as login name, teletype they logged in at, when they logged in, and from where. Using `string.split()` on the above would not be effective, since the spacing is erratic and inconsistent. The other problem is that there is a space between the month, day, and time for the login timestamps. We would probably want to keep these fields together.

You need some way to describe a pattern such as, "split on two or more spaces." This is easily done with regular expressions. In no time, we whip up the RE pattern "`\s\s+`," which does mean at least two whitespace characters. Let's create a program called `rewho.py` that reads the output of the `who` command, presumably saved into a file called `whodata.txt`. Our `rewho.py` script initially looks something like this:

```
import re
f = open('whodata.txt', 'r')
for eachLine in f.readlines():
         print re.split('\s\s+', eachLine)
f.close()
```

We will now execute the `who` command, saving the output into `whodata.txt`, and then call `rewho.py` and take a look at the results:

```
% who > whodata.txt
% rewho.py
['wesc', 'console', 'Jun 20 20:33\012']
['wesc', 'pts/9', 'Jun 22 01:38\011(192.168.0.6)\012']
['wesc', 'pts/1', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/2', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/4', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/3', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/5', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/6', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/7', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/8', 'Jun 20 20:33\011(:0.0)\012']
```

It was a good first try, but not quite correct. For one thing, we did not anticipate a single TAB (ASCII `\011`) as part of the output (which looked like at least two spaces, right?), and perhaps we aren't really keen on saving the NEWLINE (ASCII `\012`), which terminates each line. We are now going to fix those problems as well as improve the overall quality of our application by making a few more changes.

First, we would rather run the `who` command from within the script, instead of doing it externally and saving the output to a whodata.txt filedoing this repeatedly gets tiring rather quickly. To accomplish invoking another program from within ours, we call upon the os.popen() command, discussed briefly in Section 14.5.2. Although os.popen() is available only on Unix systems, the point is to illustrate the functionality of re.split(), which is available on all platforms.

We get rid of the trailing NEWLINEs and add the detection of a single TAB as an additional, alternative re.split() delimiter. Presented in Example 15.1 is the final version of our rewho.py script:

Example 15.1. Split Output of Unix who Command (rewho.py)

This script calls the who *command and parses the input by splitting up its data along various types of whitespace characters.*

```
#!/usr/bin/env python

from os import popen
from re import split

f = popen('who', 'r')
for eachLine in f.readlines():
  print split('\s\s+|\t', eachLine.strip())
f.close()
```

Running this script, we now get the following (correct) output:

```
% rewho.py
['wesc', 'console', 'Jun 20 20:33']
['wesc', 'pts/9', 'Jun 22 01:38', '(192.168.0.6)']
['wesc', 'pts/1', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/2', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/4', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/3', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/5', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/6', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/7', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/8', 'Jun 20 20:33', '(:0.0)']
```

A similar exercise can be achieved in a DOS/Windows environment using the dir command in place of who.

While the subject of ASCII characters is still warm, we would like to note that there can be confusion between regular expression special characters and special ASCII symbols. We may use \n to represent an ASCII NEWLINE character, but we may use \d meaning a regular expression match of a single numeric digit. Problems may occur if there is a symbol used by both ASCII and regular expressions, so in the Core Note on the following page, we recommend the use of Python raw strings to prevent any problems. One more caution: the "\w" and "\W" alphanumeric character sets are affected by the L or LOCALE compilation flag and in Python 1.6 and newer, by Unicode flags starting in 2.0 (U or UNICODE).

Core Note: Use of Python raw strings

You may have seen the use of raw strings in some of the examples above. Regular expressions were a strong motivation for the advent of raw strings. The reason is because of conflicts between ASCII characters and regular expression special characters. As a special symbol, "`\b`" represents the ASCII character for backspace, but "`\b`" is also a regular expression special symbol, meaning "match" on a word boundary. In order for the RE compiler to see the two characters "`\b`" as your string and not a (single) backspace, you need to escape the backslash in the string by using another backslash, resulting in "`\\b`."

This can get messy, especially if you have a lot of special characters in your string, adding to the confusion. We were introduced to raw strings back in Chapter 6, and they can be (and are often) used to help keep REs looking somewhat manageable. In fact, many Python programmers swear by these and only use raw strings when defining regular expressions.

Here are some examples of differentiating between the backspace "`\b`" and the regular expression "`\b`," with and without raw strings:

```
>>> m = re.match('\bblow', 'blow')  # backspace, no match
>>> if m is not None: m.group()
...
>>> m = re.match('\\bblow', 'blow')  # escaped \, now it works
>>> if m is not None: m.group()
...
'blow'
>>> m = re.match(r'\bblow', 'blow')  # use raw string instead
>>> if m is not None: m.group()
...
'blow'
```

You may have recalled that we had no trouble using "`\d`" in our regular expressions without using raw strings. That is because there is no ASCII equivalent special character, so the regular expression compiler already knew you meant a decimal digit.

PREV NEXT

PREV NEXT

15.4. Regular Expressions Example

We will now run through an in-depth example of the different ways of using regular expressions for string manipulation. The first step is to come up with some code that actually generates some random (but-not-so-random) data on which to operate. In Example 15.2, we present `gendata.py`, a script that generates a data set. Although this program simply displays the generated set of strings to standard output, this output may very well be redirected to a test file.

Example 15.2. Data Generator for RE Exercises (`gendata.py`)

Create random data for regular expressions practice and output the generated data to the screen.

```python
#!/usr/bin/env python

from random import randint, choice
from string import lowercase
from sys import maxint
from time import ctime

doms = ( 'com', 'edu', 'net', 'org', 'gov' )

for i in range(randint(5, 10)):
    dtint = randint(0, maxint-1)  # pick date
    dtstr = ctime(dtint)          # date string

    shorter = randint(4, 7)       # login shorter
    em = ''
    for j in range(shorter):            # generate login
    em += choice(lowercase)

    longer = randint(shorter, 12) # domain longer
    dn = ''
    for j in range(longer):             # create domain
        dn += choice(lowercase)

    print '%s::%s@%s.%s::%d-%d-%d' % (dtstr, em,
        dn, choice(doms), dtint, shorter, longer)
```

This script generates strings with three fields, delimited by a pair of colons, or a double-colon. The first field is a random (32-bit) integer, which is converted to a date (see the accompanying Core Note). The next field is a randomly generated electronic mail (e-mail) address, and the final field is a set of integers separated by a single dash (`-`).

Running this code, we get the following output (your mileage will definitely vary) and store it locally as the file `redata.txt`:

```
Thu Jul 22 19:21:19 2004::izsp@dicqdhytvhv.edu::1090549279-4-11
Sun Jul 13 22:42:11 2008::zqeu@dxaibjgkniy.com::1216014131-4-11
```

```
Sat May  5 16:36:23 1990::fclihw@alwdbzpsdg.edu::641950583-6-10
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
Thu Jun 26 19:08:59 2036::ugxfugt@jkhuqhs.net::2098145339-7-7
Tue Apr 10 01:04:45 2012::zkwaq@rpxwmtikse.com::1334045085-5-10
```

You may or may not be able to tell, but the output from this program is ripe for regular expression processing. Following our line-by-line explanation, we will implement several REs to operate on these data, as well as leave plenty for the end-of-chapter exercises.

Line-by-Line Explanation

Lines 16

In our example script, we require the use of multiple modules. But since we are utilizing only one or two functions from these modules, rather than importing the entire module, we choose in this case to import only specific attributes from these modules. Our decision to use **`from-import`** rather than **`import`** was based solely on this reasoning. The **`from-import`** lines follow the Unix startup directive on line 1.

Line 8

`doms` is simply a set of higher-level domain names from which we will randomly pick for each randomly generated e-mail address.

Lines 1012

Each time `gendata.py` executes, between 5 and 10 lines of output are generated. (Our script uses the `random.randint()` function for all cases where we desire a random integer.) For each line, we choose a random integer from the entire possible range (0 to 2^{31} - 1 [`sys.maxint`]), then convert that integer to a date using `time.ctime()`. System time in Python and most Unix-based computers is based on the number of seconds that have elapsed since the "epoch," midnight UTC/GMT on January 1, 1970. If we choose a 32-bit integer, that represents one moment in time from the epoch to the maximum possible time, 2^{32} seconds *after* the epoch.

Lines 1422

The login name for the fake e-mail address should be between 4 and 7 characters in length. To put it together, we randomly choose between 4 and 7 random lowercase letters, concatenating each letter to our string one at a time. The functionality of the `random.choice()` function is given a sequence, return a random element of that sequence. In our case, the sequence is the set of all 26 lowercase letters of the alphabet, `string.lowercase.`

We decided that the main domain name for the fake e-mail address should be between 4 and 12 characters in length, but at least as long as the login name. Again, we use random lowercase letters to put this name together letter by letter.

Lines 2425

The key component of our script puts together all of the random data into the output line. The date string comes first, followed by the delimiter. We then put together the random e-mail address by concatenating the login name, the "`@`" symbol, the domain name, and a randomly chosen high-level domain. After the final double-colon, we put together a random integer string using the original time chosen (for the date string), followed by the lengths of the login and domain names, all separated by a single hyphen.

15.4.1. Matching a String

For the following exercises, create both permissive and restrictive versions of your REs. We recommend you test these REs in a short application that utilizes our sample `redata.txt` file above (or use your own generated data from running `gendata.py`). You will need to use it again when you do the exercises.

To test the RE before putting it into our little application, we will import the `re` module and assign one sample line from `redata.txt` to a string variable `data`. These statements are constant across both illustrated examples.

```
>>> import re
>>> data = 'Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

In our first example, we will create a regular expression to extract (only) the days of the week from the timestamps from each line of the data file `redata.txt`. We will use the following RE:

```
"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"
```

This example requires that the string start with ("`^`" RE operator) any of the seven strings listed. If we were to "translate" the above RE to English, it would read something like, "the string should start with "Mon," "Tue,"..., "Sat," or "Sun."

Alternatively, we can bypass all the carat operators with a single carat if we group the day strings like this:

```
"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"
```

The parentheses around the set of strings mean that one of these strings must be encountered for a match to succeed. This is a "friendlier" version of the original RE we came up with, which did not have the parentheses. Using our modified RE, we can take advantage of the fact that we can access the matched string as a subgroup:

```
>>> patt = '^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)'
>>> m = re.match(patt, data)
>>> m.group()                    # entire match
'Thu'
>>> m.group(1)                   # subgroup 1
'Thu'
>>> m.groups()                   # all subgroups
('Thu',)
```

This feature may not seem as revolutionary as we have made it out to be for this example, but it is definitely advantageous in the next example or anywhere you provide extra data as part of the RE to help in the string matching process, even though those characters may not be part of the string you are interested in.

Both of the above REs are the most restrictive, specifically requiring a set number of strings. This may not work well in an internationalization environment where localized days and abbreviations are used. A looser RE would be: "`^\w{3}`." This one requires only that a string begin with three consecutive alphanumeric characters. Again, to translate the RE into English, the carat indicates "begins with," the "`\w`" means any single alphanumeric character, and the "`{3}`" means that there should be 3 consecutive copies of the RE which the "`{3}`" embellishes. Again, if you want grouping, parentheses should be used, i.e., "`^(\w{3})`":

```
>>> patt = '^(\w{3})'
>>> m = re.match(patt, data)
>>> if m is not None: m.group()
...
'Thu'
>>> m.group(1)
'Thu'
```

Note that an RE of "`^(\w){3}`" is not correct. When the "`{3}`" was inside the parentheses, the match for three consecutive alphanumeric characters was made first, then represented as a group. But by moving the "`{3}`" outside, it is now equivalent to three consecutive single alphanumeric characters:

```
>>> patt = '^(\w){3}'
>>> m = re.match(patt, data)
>>> if m is not None: m.group()
...
'Thu'
>>> m.group(1)
'u'
```

The reason why only the "u" shows up when accessing subgroup 1 is that subgroup 1 was being continually replaced by the next character. In other words, `m.group(1)` started out as "T," then changed to "h," then finally was replaced by "u." These are three individual (and overlapping) groups of a single alphanumeric character, as opposed to a single group consisting of three consecutive alphanumeric characters.

In our next (and final) example, we will create a regular expression to extract the numeric fields found at the end of each line of `redata.txt`.

15.4.2. Search versus Match, and Greediness too

Before we create any REs, however, we realize that these integer data items are at the end of the data strings. This means that we have a choice of using either search or match. Initiating a search makes more sense because we know exactly what we are looking for (set of three integers), that what we seek is not at the beginning of the string, and that it does not make up the entire string. If we were to perform a match, we would have to create an RE to match the entire line and use subgroups to save the data we are interested in. To illustrate the differences, we will perform a search first, then do a match to show you that searching is more appropriate.

Since we are looking for three integers delimited by hyphens, we create our RE to indicate as such: "`\d+-\d+-\d+`". This regular expression means, "any number of digits (at least one, though) followed by a hyphen, then more digits, another hyphen, and finally, a final set of digits." We test our RE now using `search()`:

```
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group()      # entire match
'1171590364-6-8'
```

A match attempt, however, would fail. Why? Because matches start at the beginning of the string, the numeric strings are at the rear. We would have to create another RE to match the entire string. We can be lazy though, by using ".+" to indicate just an arbitrary set of characters followed by what we are really interested in:

```
patt = '.+\d+-\d+-\d+'
>>> re.match(patt, data).group()       # entire match
'Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-
6-8'
```

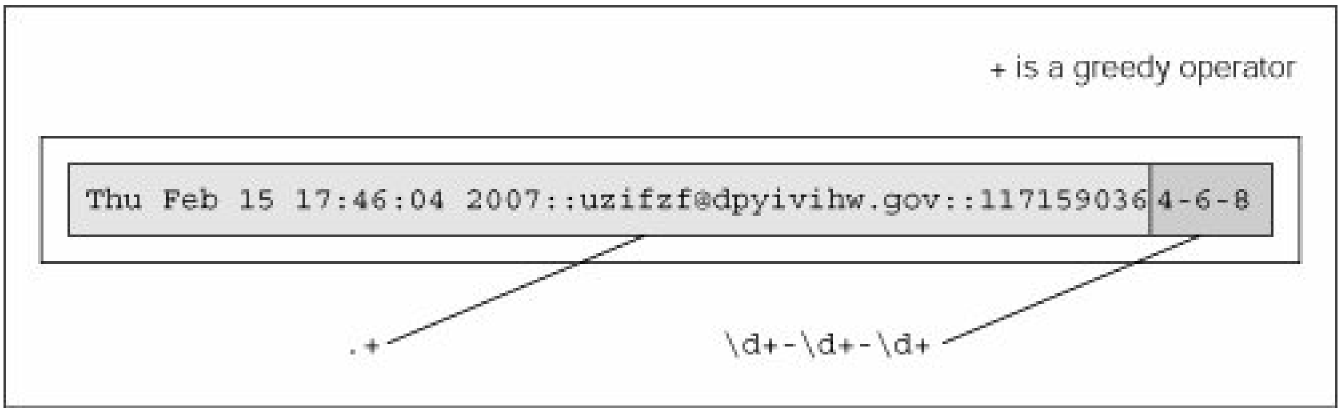
This works great, but we really want the number fields at the end, not the entire string, so we have to use parentheses to group what we want:

```
>>> patt = '.+(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)      # subgroup 1
'4-6-8'
```

What happened? We should have extracted "1171590364-6-8," not just "4-6-8." Where is the rest of the first integer? The problem is that regular expressions are inherently greedy. That means that with wildcard patterns, regular expressions are evaluated in left-to-right order and try to "grab" as many characters as possible which match the pattern. In our case above, the ".+" grabbed every single character from the beginning of the string, including most of the first integer field we wanted. The "\d+" needed only a single digit, so it got "4", while the ".+" matched everything from the beginning of the string up to that first digit: "Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::117159036", as indicated below in Figure 15-2.

Figure 15-2. Why our match went awry: + is a greedy operator

[View full size image]

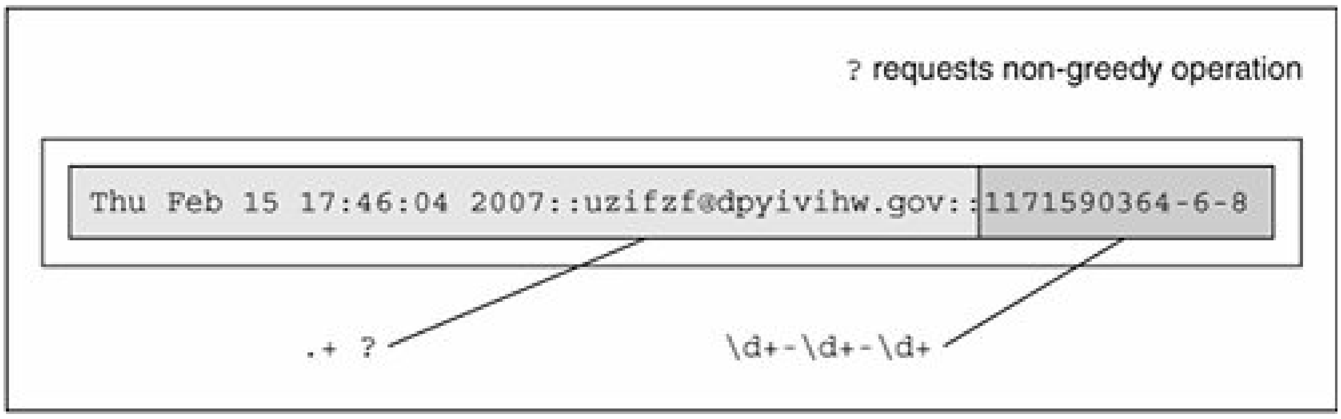


One solution is to use the "don't be greedy" operator, "?". It can be used after "*", "+", or "?". This directs the regular expression engine to match as few characters as possible. So if we place a "?" after the ".+", we obtain the desired result illustrated in Figure 15-3.

```
>>> patt = '.+?(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)         # subgroup 1
'1171590364-6-8'
```

Figure 15-3. Solving the greedy problem: ? requests non-greediness

[View full size image]


? requests non-greedy operation
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
.+ ?
\d+-\d+-\d+

Another solution, which is actually easier, is to recognize that "::" is our field separator. You can then just use the regular string `strip('::')` method and get all the parts, then take another split on the dash with `strip('-')` to obtain the three integers you were originally seeking. Now, we did not choose this solution first because this is how we put the strings together to begin with using `gendata.py`!

One final example: let us say we want to pull out only the middle integer of the three-integer field. Here is how we would do it (using a search so we don't have to match the entire string): "`-(\d+)-`". Trying out this pattern, we get:

```
>>> patt = '-(\d+)-'
>>> m = re.search(patt, data)
>>> m.group()                   # entire match
'-6-'
>>> m.group(1)                  # subgroup 1
'6'
```

We barely touched upon the power of regular expressions, and in this limited space we have not been able to do them justice. However, we hope that we have given an informative introduction so that you can add this powerful tool to your programming skills. We suggest you refer to the documentation for more details on how to use REs with Python. For more complete immersion into the world of regular expressions, we recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

PREV NEXT

15.5. Exercises

Regular Expressions. Create regular expressions in Exercises 15-1 to 15-12 to:

15-1. Recognize the following strings: "bat," "bit," "but," "hat," "hit," or "hut."

15-2. Match any pair of words separated by a single space, i.e., first and last names.

15-3. Match any word and single letter separated by a comma and single space, as in last name, first initial.

15-4. Match the set of all valid Python identifiers.

15-5. Match a street address according to your local format (keep your RE general enough to match any number of street words, including the type designation). For example, American street addresses use the format: 1180 Bordeaux Drive. Make your RE general enough to support multi-word street names like: 3120 De la Cruz Boulevard.

15-6. Match simple Web domain names that begin with "www." and end with a ".com" suffix, e.g., www.yahoo.com. Extra credit if your RE also supports other high-level domain names: .edu, .net, etc., e.g., www.ucsc.edu.

15-7. Match the set of the string representations of all Python integers.

15-8. Match the set of the string representations of all Python longs.

15-9. Match the set of the string representations of all Python floats.

15-10. Match the set of the string representations of all Python complex numbers.

15-11. Match the set of all valid e-mail addresses (start with a loose RE, then try to tighten it as much as you can, yet maintain correct functionality).

15-12. Match the set of all valid Web site addresses (URLs) (start with a loose RE, then try to tighten it as much as you can, yet maintain correct functionality).

15-13. *type()*. The `type()` built-in function returns a type object, which is displayed as a Pythonic-looking string:

```
>>> type(0)
<type
'int'>
>>> type(.34)
<type
'float'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

Create an RE that would extract out the actual type name from the string. Your function should take a string like this "`<type 'int'>`" and return "`int`". (Ditto for all other types, i.e., 'float', 'builtin_function_or_method', etc.) Note: You are implementing the value that is stored in the `__name__` attribute for classes and some built-in types.

15-14. *Regular Expressions*. In Section 15.2, we gave you the RE pattern that matched the single- or double-digit string representations of the months January to September ("`0?[1-9]`"). Create the RE that represents the remaining three months in the standard calendar.

15-15. *Regular Expressions*. Also in Section 15.2, we gave you the RE pattern that matched credit card (CC) numbers ("`[0-9]{15,16}`"). However, this pattern does not allow for hyphens separating blocks of numbers. Create the RE that allows hyphens, but only in the correct locations. For example, 15-digit CC numbers have a pattern of 4-6-5, indicating four digits-hyphen-six digits-hyphen-five digits, and 16-digit CC numbers have a 4-4-4-4 pattern. Remember to "balloon" the size of the entire string correctly. Extra credit: There is a standard algorithm for determining whether a CC number is valid. Write some code not only to recognize a correctly formatted CC number, but also a valid one.

The next set of problems (15-16 through 15-27) deal specifically with the data that are generated by `gendata.py`. Before approaching problems 15-17 and 15-18, you may wish to do 15-16 and all the regular expressions first.

15-16. Update the code for `gendata.py` so that the data are written directly to `redata.txt` rather than output to the screen.

15-17. Determine how many times each day of the week shows up for any incarnation of `redata.txt`. (Alternatively, you can also count how many times each month of the year was chosen.)

15-18. Ensure there is no data corruption in `redata.txt` by confirming that the first integer of the integer field matches the timestamp given at the front of each output line.

Create regular expressions to:

15-19. Extract the complete timestamps from each line.

15-20. Extract the complete e-mail address from each line.

15-21. Extract only the months from the timestamps.

15-22. Extract only the years from the timestamps.

15-23. Extract only the time (HH:MM:SS) from the timestamps.

15-24. Extract only the login and domain names (both the main domain name and the high-level domain together) from the e-mail address.

15-25. Extract only the login and domain names (both the main domain name and the high-level domain) from the e-mail address.

15-26. Replace the e-mail address from each line of data with your e-mail address.

15-27. Extract the months, days, and years from the timestamps and output them in "Mon Day, Year" format, iterating over each line only once.

For problems 15-28 and 15-29, recall the regular expression introduced in Section 15.2, which matched telephone numbers but allowed for an optional area code prefix: `\d{3}-\d{3}-\d{4}`*. Update this regular expression so that:*

15-28. Area codes (the first set of three-digits and the accompanying hyphen) are optional, i. e., your RE should match both 800-555-1212 as well as just 555-1212.

15-29. Either parenthesized or hyphenated area codes are supported, not to mention optional; make your RE match 800-555-1212, 555-1212, and also (800) 555-1212.

PREV NEXT

PREV NEXT

Chapter 18. Multithreaded Programming

Chapter Topics

In this section, we will explore the different ways you can achieve more parallelism in your code by using the multithreaded (MT) programming features found in Python. We will begin by differentiating between processes and threads in the first few of sections of this chapter. We will then introduce the notion of multithreaded programming. (Those of you already familiar with MT programming can skip directly to Section 18.3.5.) The final sections of this chapter present some examples of how to use the `threading` and `Queue` modules to accomplish MT programming with Python.

PREV NEXT

18.1. Introduction/Motivation

Before the advent of multithreaded (MT) programming, running of computer programs consisted of a single sequence of steps that were executed in synchronous order by the host's central processing unit (CPU). This style of execution was the norm whether the task itself required the sequential ordering of steps or if the entire program was actually an aggregation of multiple subtasks. What if these subtasks were independent, having no *causal* relationship (meaning that results of subtasks do not affect other subtask outcomes)? Is it not logical, then, to want to run these independent tasks all at the same time? Such parallel processing could significantly improve the performance of the overall task. This is what MT programming is all about.

MT programming is ideal for programming tasks that are asynchronous in nature, require multiple concurrent activities, and where the processing of each activity may be *nondeterministic*, i.e., random and unpredictable. Such programming tasks can be organized or partitioned into multiple streams of execution where each has a specific task to accomplish. Depending on the application, these subtasks may calculate intermediate results that could be merged into a final piece of output.

While CPU-bound tasks may be fairly straightforward to divide into subtasks and executed sequentially or in a multithreaded manner, the task of managing a single-threaded process with multiple external sources of input is not as trivial. To achieve such a programming task without multithreading, a sequential program must use one or more timers and implement a multiplexing scheme.

A sequential program will need to sample each I/O (input/output) terminal channel to check for user input; however, it is important that the program does not block when reading the I/O terminal channel because the arrival of user input is nondeterministic, and blocking would prevent processing of other I/O channels. The sequential program must use non-blocked I/O or blocked I/O with a timer (so that blocking is only temporary).

Because the sequential program is a single thread of execution, it must juggle the multiple tasks that it needs to perform, making sure that it does not spend too much time on any one task, and it must ensure that user response time is appropriately distributed. The use of a sequential program for this type of task often results in a complicated flow of control that is difficult to understand and maintain.

Using an MT program with a shared data structure such as a `Queue` (a multithreaded queue data structure discussed later in this chapter), this programming task can be organized with a few threads that have specific functions to perform:

- `UserRequestThread:` Responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.
- `RequestProcessor:` A thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.
- `ReplyThread:` Responsible for taking output destined for the user and either sending it back, if in a networked application, or writing data to the local file system or database.

Organizing this programming task with multiple threads reduces the complexity of the program and enables an implementation that is clean, efficient, and well organized. The logic in each thread is typically less complex because it has a specific job to do. For example, the `UserRequestThread` simply reads input from a user and places the data into a queue for further processing by another thread, etc. Each thread has its own job to do; you merely have to design each type of thread to do one thing and do it well. Use of threads for specific tasks is not unlike Henry Ford's assembly line model for manufacturing automobiles.

18.2. Threads and Processes

18.2.1. What Are Processes?

Computer *programs* are merely executables, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A *process* (sometimes called a *heavyweight process*) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes. Processes can also *fork* or *spawn* new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

18.2.2. What Are Threads?

Threads (sometimes called *lightweight processes*) are similar to processes except that they all execute within the same process, and thus all share the same context. They can be thought of as "mini-processes" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be preempted (interrupted) and temporarily put on hold (also known as *sleeping*) while other threads are runningthis is called *yielding*.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with one another more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial "back of the line" to await more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a *race condition*. Fortunately, most thread libraries come with some sort of synchronization primitives that allow the thread manager to control execution and access.

Another caveat is that threads may not be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.

18.3. Python, Threads, and the Global Interpreter Lock

18.3.1. Global Interpreter Lock (GIL)

Execution of Python code is controlled by the *Python Virtual Machine* (aka the interpreter main loop). Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only *one* is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by the *global interpreter lock* (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

1. Set the GIL
2. Switch in a thread to run
3. Execute either ...
   a. For a specified number of bytecode instructions, or
   b. If the thread voluntarily yields control (can be accomplished `time.sleep(0)`)
4. Put the thread back to sleep (switch out thread)
5. Unlock the GIL, and ...
6. Do it all over again (lather, rinse, repeat)

When a call is made to external code, i.e., any C/C++ extension built-in function, the GIL will be locked until it has completed (since there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL, however, so you being the Python developer shouldn't have to worry about your Python code locking up in those situations.

As an example, for any Python I/O-oriented routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code that *doesn't* have much I/O will tend to keep the processor (and GIL) for the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

Those of you interested in the source code, the interpreter main loop, and the GIL can take a look at the `Python/ceval.c` file.

18.3.2. Exiting Threads

When a thread completes execution of the function it was created for, it exits. Threads may also quit by calling an exit function such as `tHRead`. `exit()`, or any of the standard ways of exiting a Python process, i. e., `sys.exit()` or raising the `SystemExit` exception. You cannot, however, go and "kill" a thread.

We will discuss in detail the two Python modules related to threads in the next section, but of the two, the `thread` module is the one we do *not* recommend. There are many reasons for this, but an obvious one is that when the main thread exits, all other threads die without cleanup. The other module, `threading`, ensures that the whole process stays alive until all "important" child threads have exited. (We will clarify what "important" means soon. Look for the *daemon threads* Core Tip sidebar.)

Main threads should always be good managers, though, and perform the task of knowing what needs to be executed by individual threads, what data or arguments each of the spawned threads requires, when they complete execution, and what results they provide. In so doing, those main threads can collate the individual results into a final, meaningful conclusion.

18.3.3. Accessing Threads from Python

Python supports multithreaded programming, depending on the operating system that it is running on. It is supported on most Unix-based platforms, i.e., Linux, Solaris, MacOS X, *BSD, as well as Win32 systems. Python uses POSIX-compliant threads, or "pthreads," as they are commonly known.

By default, threads are enabled when building Python from source (since Python 2.0) or the Win32 installed binary. To tell whether threads are available for your interpreter, simply attempt to import the `thread` module from the interactive interpreter. No errors occur when threads are available:

```
>>> import thread
>>>
```

If your Python interpreter was *not* compiled with threads enabled, the module import fails:

```
>>> import thread
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named thread
```

In such cases, you may have to recompile your Python interpreter to get access to threads. This usually involves invoking the `configure` script with the "`--with-thread`" option. Check the `README` file for your distribution to obtain specific instructions on how to compile Python with threads for your system.

18.3.4. Life Without Threads

For our first set of examples, we are going to use the `time.sleep()` function to show how threads work. `time.sleep()` takes a floating point argument and "sleeps" for the given number of seconds, meaning that execution is temporarily halted for the amount of time specified.

Let us create two "time loops," one that sleeps for 4 seconds and one that sleeps for 2 seconds, `loop0()` and `loop1()`, respectively. (We use the names "loop0" and "loop1" as a hint that we will eventually have a sequence of loops.) If we were to execute `loop0()` and `loop1()` sequentially in a one-process or single-threaded program, as `onethr.py` does in Example 18.1, the total execution time would be at least 6 seconds. There may or may not be a 1-second gap between the starting of `loop0()` and `loop1()`, and other execution overhead which may cause the overall time to be bumped to 7 seconds.

Example 18.1. Loops Executed by a Single Thread (`onethr.py`)

Executes two loops consecutively in a single-threaded program. One loop must complete before the other can begin. The total elapsed time is the sum of times taken by each loop.

```
#!/usr/bin/env python

from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
    print 'loop 0 done at:', ctime()

def loop1():
    print 'start loop 1 at:', ctime()
    sleep(2)
    print 'loop 1 done at:', ctime()

def main():
    print 'starting at:', ctime()
    loop0()
    loop1()
    print 'all DONE       at:', ctime()

if __name__ == '__main__':
    main()
```

We can verify this by executing `onethr.py`, which gives the following output:

```
$ onethr.py
starting at: Sun Aug 13 05:03:34 2006
start loop 0 at: Sun Aug 13 05:03:34 2006
loop 0 done at: Sun Aug 13 05:03:38 2006
start loop 1 at: Sun Aug 13 05:03:38 2006
loop 1 done at: Sun Aug 13 05:03:40 2006
all DONE at: Sun Aug 13 05:03:40 2006
```

Now, pretend that rather than sleeping, `loop0()` and `loop1()` were separate functions that performed individual and independent computations, all working to arrive at a common solution. Wouldn't it be useful to have them run in parallel to cut down on the overall running time? That is the premise behind MT that we now introduce to you.

18.3.5. Python Threading Modules

Python provides several modules to support MT programming, including the `tHRead`, `tHReading`, and `Queue` modules. The `thread` and `threading` modules allow the programmer to create and manage threads. The `thread` module provides basic thread and locking support, while `threading` provides higher-level, fully featured thread management. The `Queue` module allows the user to create a queue data structure that can be shared across multiple threads. We will take a look at these modules individually and present examples and intermediate-sized applications.

Core Tip: Avoid use of `thread` module

We recommend avoiding the `tHRead` module for many reasons. The first is that the high-level `threading` module is more contemporary, not to mention the fact that thread support in the `threading` module is much improved and the use of attributes of the `thread` module may conflict with using the `threading` module. Another reason is that the lower-level `thread` module has few synchronization primitives (actually only one) while `threading` has many.

However, in the interest of learning Python and threading in general, we do present some code that uses the `thread` module. These pieces of code should be used for learning purposes only and will give you a much better insight as to why you would want to avoid using the `thread` module. These examples also show how our applications and thread programming improve as we migrate to using more appropriate tools such as those available in the `threading` and `Queue` modules.

Another reason to avoid using `thread` is because there is no control of when your process exits. When the main thread finishes, all threads will also die, without warning or proper cleanup. As mentioned earlier, at least `tHReading` allows the important child threads to finish first before exiting.

Use of the `tHRead` module is recommended only for experts desiring lower-level thread access. Those of you new to threads should look at the code samples to see how we can overlay threads onto our time loop application and to gain a better understanding as to how these first examples evolve to the main code samples of this chapter. Your first multithreaded application should utilize `threading` and perhaps other high-level thread modules, if applicable.

PREV NEXT

PREV NEXT

18.4. tHRead Module

Let's take a look at what the tHRead module has to offer. In addition to being able to spawn threads, the tHRead module also provides a basic synchronization data structure called a *lock object* (aka primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore). As we mentioned earlier, such synchronization primitives go hand in hand with thread management.

Listed in Table 18.1 are the more commonly used thread functions and LockType lock object methods.

Table 18.1. thread Module and Lock Objects

Function/Method	***Description***
tHRead ***Module Functions***	
start_new_thread(*function, args, kwargs=None*)	Spawns a new thread and execute *function* with the given *args* and optional *kwargs*
allocate_lock()	Allocates LockType lock object
exit()	Instructs a thread to exit
LockType ***Lock Object Methods***	
acquire(*wait*=None)	Attempts to acquire lock object
locked()	Returns True if lock acquired, False otherwise
release()	Releases lock

The key function of the thread module is start_new_thread(). Its syntax is exactly that of the apply() built-in function, taking a function along with arguments and optional keyword arguments. The difference is that instead of the main thread executing the function, a new thread is spawned to invoke the function.

Let's take our onethr.py example and integrate threading into it. By slightly changing the call to the loop*() functions, we now present mtsleep1.py in Example 18.2.

Example 18.2. Using the thread Module (mtsleep1.py)

The same loops from onethr.py are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

```
#!/usr/bin/env python

import thread
from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
    print 'loop 0 done at:', ctime()

def loop1():
    print 'start loop 1 at:', ctime()
    sleep(2)
    print 'loop 1 done at:', ctime()

def main():
    print 'starting at:', ctime()
    thread.start_new_thread(loop0, ())
    thread.start_new_thread(loop1, ())
    sleep(6)
    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

`start_new_thread()` requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4, the length of time of our longest loop, plus any overhead.

```
$ mtsleep1.py
starting at: Sun Aug 13 05:04:50 2006
start loop 0 at: Sun Aug 13 05:04:50 2006
start loop 1 at: Sun Aug 13 05:04:50 2006
loop 1 done at: Sun Aug 13 05:04:52 2006
loop 0 done at: Sun Aug 13 05:04:54 2006
all DONE at: Sun Aug 13 05:04:56 2006
```

The pieces of code that sleep for 4 and 2 seconds now occur concurrently, contributing to the lower overall runtime. You can even see how loop 1 finishes before loop 0.

The only other major change to our application is the addition of the "`sleep(6)`" call. Why is this necessary? The reason is that if we did not stop the main thread from continuing, it would proceed to the next statement, displaying "all done" and exit, killing both threads running `loop0()` and `loop1()`.

We did not have any code that told the main thread to wait for the child threads to complete before continuing. This is what we mean by threads requiring some sort of synchronization. In our case, we used another `sleep()` call as our synchronization mechanism. We used a value of 6 seconds because we know that both threads (which take 4 and 2 seconds, as you know) should have completed by the time the main thread has counted to 6.

You are probably thinking that there should be a better way of managing threads than creating that extra delay of 6 seconds in the main thread. Because of this delay, the overall runtime is no better than in our single-threaded version. Using `sleep()` for thread synchronization as we did is not reliable. What if our loops had independent and varying execution times? We may be exiting the main thread too early or too late. This is where locks come in.

Making yet another update to our code to include locks as well as getting rid of separate loop functions, we get `mtsleep2.py`, presented in Example 18.3. Running it, we see that the output is similar to `mtsleep1.py`. The only difference is that we did not have to wait the extra time for `mtsleep1.py` to conclude. By using locks, we were able to exit as soon as both threads had completed execution.

```
$ mtsleep2.py
starting at: Sun Aug 13 16:34:41 2006
start loop 0 at: Sun Aug 13 16:34:41 2006
start loop 1 at: Sun Aug 13 16:34:41 2006
loop 1 done at: Sun Aug 13 16:34:43 2006
loop 0 done at: Sun Aug 13 16:34:45 2006
all DONE at: Sun Aug 13 16:34:45 2006
```

Example 18.3. Using `tHRead` and Locks (`mtsleep2.py`)

Rather than using a call to sleep() to hold up the main thread as in mtsleep1.py, the use of locks makes more sense.

```
#!/usr/bin/env python

import thread
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec, lock):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()
    lock.release()

def main():
    print 'starting at:', ctime()
    locks = []
    nloops = range(len(loops))

    for i in nloops:
        lock = thread.allocate_lock()
        lock.acquire()
        locks.append(lock)

    for i in nloops:
```

```
25            thread.start_new_thread(loop,
26                (i, loops[i], locks[i]))
27
28        for i in nloops:
29            while locks[i].locked(): pass
30
31        print 'all DONE at:', ctime()
32
33    if __name__ == '__main__':
34        main()
```

So how did we accomplish our task with locks? Let us take a look at the source code.

Line-by-Line Explanation

Lines 16

After the Unix startup line, we import the `thread` module and a few familiar attributes of the `time` module. Rather than hardcoding separate functions to count to 4 and 2 seconds, we will use a single `loop()` function and place these constants in a list, `loops`.

Lines 812

The `loop()` function will proxy for the now-removed `loop*()` functions from our earlier examples. We had to make some cosmetic changes to `loop()` so that it can now perform its duties using locks. The obvious changes are that we need to be told which loop number we are as well as how long to sleep for. The last piece of new information is the lock itself. Each thread will be allocated an acquired lock. When the `sleep()` time has concluded, we will release the corresponding lock, indicating to the main thread that this thread has completed.

Lines 1434

The bulk of the work is done here in `main()` using three separate **`for`** loops. We first create a list of locks, which we obtain using the `thread.allocate_lock()` function and acquire (each lock) with the `acquire()` method. Acquiring a lock has the effect of "locking the lock." Once it is locked, we add the lock to the lock list, `locks`. The next loop actually spawns the threads, invoking the `loop()` function per thread, and for each thread, provides it with the loop number, the time to sleep for, and the acquired lock for that thread. So why didn't we start the threads in the lock acquisition loop? There are several reasons: (1) we wanted to synchronize the threads, so that "all the horses started out the gate" around the same time, and (2) locks take a little bit of time to be acquired. If your thread executes "too fast," it is possible that it completes before the lock has a chance to be acquired.

It is up to each thread to unlock its lock object when it has completed execution. The final loop just sits and spins (pausing the main thread) until both locks have been released before continuing execution. Since we are checking each lock sequentially, we may be at the mercy of all the slower loops if they are more toward the beginning of the set of loops. In such cases, the majority of the wait time may be for the first loop(s). When that lock is released, remaining locks may have already been unlocked (meaning that corresponding threads have completed execution). The result is that the main thread will fly through those lock checks without pause. Finally, you should be well aware that the final pair of lines will execute `main()` only if we are invoking this script directly.

As hinted in the earlier Core Note, we presented the `tHRead` module only to introduce the reader to threaded programming. Your MT application should use higher-level modules such as the `threading` module, which we will now discuss.

PREV NEXT

18.5. tHReading Module

We will now introduce the higher-level tHReading module, which gives you not only a THRead class but also a wide variety of synchronization mechanisms to use to your heart's content. Table 18.2 represents a list of all the objects available in the tHReading module.

Table 18.2. threading Module Objects

tHReading *Module Objects*	*Description*
Thread	Object that represents a single thread of execution
Lock	Primitive lock object (same lock object as in the tHRead module)
RLock	Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
Condition	Condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value EventGeneral version of condition variables whereby any number of threads are waiting for some event to occur and all will awaken when the event happens
Semaphore	Provides a "waiting area"-like structure for threads waiting on a lock
BoundedSemaphore	Similar to a Semaphore but ensures it never exceeds its initial value
Timer	Similar to Thread except that it waits for an allotted period of time before running

In this section, we will examine how to use the THRead class to implement threading. Since we have already covered the basics of locking, we will not cover the locking primitives here. The THRead() class also contains a form of synchronization, so explicit use of locking primitives is not necessary.

Core Tip: Daemon threads

Another reason to avoid using the `thread` module is that it does not support the concept of daemon (or daemonic) threads. When the main thread exits, all child threads will be killed regardless of whether they are doing work. The concept of daemon threads comes into play here if you do not want this behavior.

Support for daemon threads is available in the `threading` module, and here is how they work: a daemon is typically a server that waits for client requests to service. If there is no client work to be done, the daemon just sits around idle. If you set the daemon flag for a thread, you are basically saying that it is non-critical, and it is okay for the process to exit without waiting for it to "finish." As you have seen in Chapter 16, "Network Programming" server threads run in an infinite loop and do not exit in normal situations.

If your main thread is ready to exit and you do not care to wait for the child threads to finish, then set their daemon flag. Think of setting this flag as denoting a thread to be "not important." You do this by calling each thread's `setDaemon()` method, e.g., `thread.setDae-mon(True)`, before it begins running (`tHRead.start()`.)

If you want to wait for child threads to finish, just leave them as-is, or ensure that their daemon flags are off by explicitly calling `tHRead. setDaemon (False)` before starting them. You can check a thread's daemonic status with `thread.isDaemon()`. A new child thread inherits its daemonic flag from its parent. The entire Python program will stay alive until all non-daemonic threads have exited, in other words, when no active non-daemonic threads are left).

18.5.1. Thread Class

The `THRead` class of the `threading` is your primary executive object. It has a variety of functions not available to the `thread` module, and are outlined in Table 18.3.

Table 18.3. Thread Object Methods

Method	***Description***
`start()`	Begin thread execution
`run()`	Method defining thread functionality (usually overridden by application writer in a subclass)
`join(`*`timeout`*` = None)`	Suspend until the started thread terminates; blocks unless *`timeout`* (in seconds) is given
`getName()`	Return name of thread
`setName`*`(name)`*	Set name of thread
`isAlive()`	Boolean flag indicating whether thread is still running

`isDaemon()`	Return daemon flag of thread
`setDaemon(daemonic)`	Set the daemon flag of thread as per the Boolean *daemonic* (must be called before thread `start()`ed)

There are a variety of ways you can create threads using the `Thread` class. We cover three of them here, all quite similar. Pick the one you feel most comfortable with, not to mention the most appropriate for your application and future scalability (we like the final choice the best):

- Create `Thread` instance, passing in function
- Create `THRead` instance, passing in callable class instance
- Subclass `THRead` and create subclass instance

Create `THRead` Instance, Passing in Function

In our first example, we will just instantiate `THRead`, passing in our function (and its arguments) in a manner similar to our previous examples. This function is what will be executed when we direct the thread to begin execution. Taking our `mtsleep2.py` script and tweaking it, adding the use of `Thread` objects, we have `mtsleep3.py`, shown in Example 18.4.

Example 18.4. Using the `tHReading` Module (`mtsleep3.py`)

The Thread class from the threading module has a join() method that lets the main thread wait for thread completion.

```
#!/usr/bin/env python

import threading
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))

   for i in nloops:
       t = threading.Thread(target=loop,
           args=(i, loops[i]))
       threads.append(t)

   for i in nloops:           # start threads
       threads[i].start()

   for i in nloops:           # wait for all
       threads[i].join()      # threads to finish

    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

When we run it, we see output similar to its predecessors' output:

```
$ mtsleep3.py
starting at: Sun Aug 13 18:16:38 2006
start loop 0 at: Sun Aug 13 18:16:38 2006
start loop 1 at: Sun Aug 13 18:16:38 2006
loop 1 done at: Sun Aug 13 18:16:40 2006
loop 0 done at: Sun Aug 13 18:16:42 2006
all DONE at: Sun Aug 13 18:16:42 2006
```

So what *did* change? Gone are the locks that we had to implement when using the `tHRead` module. Instead, we create a set of `Thread` objects. When each `Thread` is instantiated, we dutifully pass in the function (`target`) and arguments (`args`) and receive a `THRead` instance in return. The biggest difference between instantiating `Thread` [calling `Thread()`] and invoking `thread.start_new_thread()` is that the new thread does not begin execution right away. This is a useful synchronization feature, especially when you don't want the threads to start immediately.

Once all the threads have been allocated, we let them go off to the races by invoking each thread's `start()` method, but not a moment before that. And rather than having to manage a set of locks (allocating, acquiring, releasing, checking lock state, etc.), we simply call the `join()` method for each thread. `join()` will wait until a thread terminates, or, if provided, a timeout occurs. Use of `join()` appears much cleaner than an infinite loop waiting for locks to be released (causing these locks to sometimes be known as "spin locks").

One other important aspect of `join()` is that it does not need to be called at all. Once threads are started, they will execute until their given function completes, whereby they will exit. If your main thread has things to do other than wait for threads to complete (such as other processing or waiting for new client requests), it should by all means do so. `join()` is useful only when you *want* to wait for thread completion.

Create `Thread` Instance, Passing in Callable Class Instance

A similar offshoot to passing in a function when creating a thread is to have a callable class and passing in an instance for executionthis is the more OO approach to MT programming. Such a callable class embodies an execution environment that is much more flexible than a function or choosing from a set of functions. You now have the power of a class object behind you, as opposed to a single function or a list/tuple of functions.

Adding our new class `ThreadFunc` to the code and making other slight modifications to `mtsleep3.py`, we get `mtsleep4.py`, given in Example 18.5.

Example 18.5. Using Callable classes (`mtsleep4.py`)

In this example we pass in a callable class (instance) as opposed to just a function. It presents more of an OO approach than mtsleep3.py.

```
#!/usr/bin/env python

import threading
from time import sleep, ctime

loops = [4,2]

class ThreadFunc(object):

    def __init__(self, func, args, name=''):
        self.name = name
        self.func = func
        self.args = args

    def __call__(self):
        apply(self.func, self.args)

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops: # create all threads
        t = threading.Thread(
            target=ThreadFunc(loop, (i, loops[i]),
            loop.__name__))
        threads.append(t)

    for i in nloops: # start all threads
        threads[i].start()

    for i in nloops: # wait for completion
        threads[i].join()

    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

If we run `mtsleep4.py`, we get the expected output:

```
$ mtsleep4.py
starting at: Sun Aug 13 18:49:17 2006
start loop 0 at: Sun Aug 13 18:49:17 2006
start loop 1 at: Sun Aug 13 18:49:17 2006
loop 1 done at: Sun Aug 13 18:49:19 2006
loop 0 done at: Sun Aug 13 18:49:21 2006
all DONE at: Sun Aug 13 18:49:21 2006
```

So what are the changes this time? The addition of the `ThreadFunc` class and a minor change to instantiate the `THRead` object, which also instantiates `THReadFunc`, our callable class. In effect, we have a double instantiation going on here. Let's take a closer look at our `THReadFunc` class.

We want to make this class general enough to use with functions other than our `loop()` function, so we added some new infrastructure, such as having this class hold the arguments for the function, the function itself, and also a function name string. The constructor `__init__()` just sets all the values.

When the `Thread` code calls our `ThreadFunc`object when a new thread is created, it will invoke the `__call__()` special method. Because we already have our set of arguments, we do not need to pass it to the `THRead()` constructor, but do have to use `apply()` in our code now because we have an argument tuple. Those of you who have Python 1.6 and higher can use the new function invocation syntax described in Section 11.6.3 instead of using `apply()` on line 16:

```
self.res = self.func(*self.args)
```

Subclass `THRead` and Create Subclass Instance

The final introductory example involves subclassing `THRead()`, which turns out to be extremely similar to creating a callable class as in the previous example. Subclassing is a bit easier to read when you are creating your threads (lines 29-30). We will present the code for `mtsleep5.py` in Example 18.6 as well as the output obtained from its execution, and leave it as an exercise for the reader to compare `mtsleep5.py` to `mtsleep4.py`.

Example 18.6. Subclassing `Thread` (`mtsleep5.py`)

Rather than instantiating the Thread class, we subclass it. This gives us more flexibility in customizing our threading objects and simplifies the thread creation call.

```
1     #!/usr/bin/env python
2
3     import threading
4     from time import sleep, ctime
5
6     loops = (4, 2)
7
8     class MyThread(threading.Thread):
9         def __init__(self, func, args, name=''):
10            threading.Thread.__init__(self)
11            self.name = name
12            self.func = func
13            self.args = args
14
15        def run(self):
16            apply(self.func, self.args)
17
18    def loop(nloop, nsec):
```

```
19          print 'start loop', nloop, 'at:', ctime()
20          sleep(nsec)
21          print 'loop', nloop, 'done at:', ctime()
22
23      def main():
24          print 'starting at:', ctime()
25          threads = []
26          nloops = range(len(loops))
27
28          for i in nloops:
29              t = MyThread(loop, (i, loops[i]),
30                  loop.__name__)
31              threads.append(t)
32
33          for i in nloops:
34              threads[i].start()
35
36          for i in nloops:
37              threads[i].join()
38
39          print 'all DONE at:', ctime()'
40
41      if __name__ == '__main__':
42          main()
```

Here is the output for `mtsleep5.py`, again, just what we expected:

```
$ mtsleep5.py
starting at: Sun Aug 13 19:14:26 2006
start loop 0 at: Sun Aug 13 19:14:26 2006
start loop 1 at: Sun Aug 13 19:14:26 2006
loop 1 done at: Sun Aug 13 19:14:28 2006
loop 0 done at: Sun Aug 13 19:14:30 2006
all DONE at: Sun Aug 13 19:14:30 2006
```

While the reader compares the source between the `mtsleep4` and `mtsleep5` modules, we want to point out the most significant changes: (1) our `MyThread` subclass constructor must first invoke the base class constructor (line 9), and (2) the former special method `__call__()` must be called `run()` in the subclass.

We now modify our `MyThread` class with some diagnostic output and store it in a separate module called `myThread` (see Example 18.7) and import this class for the upcoming examples. Rather than simply calling `apply()` to run our functions, we also save the result to instance attribute `self.res`, and create a new method to retrieve that value, `getresult()`.

Example 18.7. `MyThread` Subclass of Thread (`myThread.py`)

To generalize our subclass of Thread *from* mtsleep5.py, *we move the subclass to a separate module and add a* getResult() *method for callables that produce return values.*

```
#!/usr/bin/env python

import threading
from time import ctime

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
          threading.Thread.__init__(self)
          self.name = name
          self.func = func
          self.args = args

     def getResult(self):
         return self.res

     def run(self):
         print 'starting', self.name, 'at:', \
             ctime()
          self.res = apply(self.func, self.args)
         print self.name, 'finished at:', \
             ctime()
```

18.5.4. Fibonacci and Factorial ... Take Two, Plus Summation

The `mtfacfib.py` script, given in Example 18.8, compares execution of the recursive Fibonacci, factorial, and summation functions. This script runs all three functions in a single-threaded manner, then performs the same task using threads to illustrate one of the advantages of having a threading environment.

Example 18.8. Fibonacci, Factorial, Summation (`mtfacfib.py`)

In this MT application, we execute three separate recursive functionsfirst in a single-threaded fashion, followed by the alternative with multiple threads.

```
#!/usr/bin/env python

from myThread import MyThread
from time import ctime, sleep

def fib(x):
    sleep(0.005)
    if x < 2: return 1
    return (fib(x-2) + fib(x-1))

def fac(x):
    sleep(0.1)
    if x < 2: return 1
    return (x * fac(x-1))

```

```
def sum(x):
    sleep(0.1)
    if x < 2: return 1
    return (x + sum(x-1))

funcs = [fib, fac, sum]
n = 12

def main():
    nfuncs = range(len(funcs))

    print '*** SINGLE THREAD'
    for i in   nfuncs:
        print 'starting', funcs[i].__name__, 'at:', \
            ctime()
        print funcs[i](n)
        print funcs[i].__name__, 'finished at:', \
            ctime()

    print '\n*** MULTIPLE THREADS'
    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (n,),
            funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()
        print threads[i].getResult()

    print 'all DONE'

if __name__ == '__main__':
    main()
```

Running in single-threaded mode simply involves calling the functions one at a time and displaying the corresponding results right after the function call.

When running in multithreaded mode, we do not display the result right away. Because we want to keep our `MyThread` class as general as possible (being able to execute callables that do and do not produce output), we wait until the end to call the `geTResult()` method to finally show you the return values of each function call.

Because these functions execute so quickly (well, maybe except for the Fibonacci function), you will notice that we had to add calls to `sleep()` to each function to slow things down so that we can see how threading may improve performance, if indeed the actual work had varying execution times you certainly wouldn't pad your work with calls to `sleep()`. Anyway, here is the output:

```
$ mtfacfib.py
*** SINGLE THREAD
starting fib at: Sun Jun 18 19:52:20 2006
```

```
233
fib finished at: Sun Jun 18 19:52:24 2006
starting fac at: Sun Jun 18 19:52:24 2006
479001600
fac finished at: Sun Jun 18 19:52:26 2006
starting sum at: Sun Jun 18 19:52:26 2006
78
sum finished at: Sun Jun 18 19:52:27 2006

*** MULTIPLE THREADS
starting fib at: Sun Jun 18 19:52:27 2006
starting fac at: Sun Jun 18 19:52:27 2006
starting sum at: Sun Jun 18 19:52:27 2006
fac finished at: Sun Jun 18 19:52:28 2006
sum finished at: Sun Jun 18 19:52:28 2006
fib finished at: Sun Jun 18 19:52:31 2006
233
479001600
78
all DONE
```

18.5.5. Other `Threading` Module Functions

In addition to the various synchronization and threading objects, the `Threading` module also has some supporting functions, detailed in Table 18.4.

Table 18.4. `threading` Module Functions

Function	*Description*
`activeCount()`	Number of currently active `Thread` objects
`currentThread()`	Returns the current `THRead` object
`enumerate()`	Returns list of all currently active `Threads`
`settrace(`*`func`*`)`[a]	Sets a trace *function* for all threads
`setprofile(`*`func`*`)`[a]	Sets a profile *function* for all threads

[a] New in Python 2.3.

18.5.6. Producer-Consumer Problem and the `Queue` Module

The final example illustrates the producer-consumer scenario where a producer of goods or services creates goods and places it in a data structure such as a queue. The amount of time between producing goods is non-deterministic, as is the consumer consuming the goods produced by the producer.

We use the `Queue` module to provide an interthread communication mechanism that allows threads to share data with each other. In particular, we create a queue into which the producer (thread) places new goods and the consumer (thread) consumes them. To do this, we will use the following attributes from the `Queue` module (see Table 18.5).

Table 18.5. Common `Queue` Module Attributes

Function/Method	***Description***
`Queue` ***Module Function***	
`queue(size)`	Creates a `Queue` object of given `size`
`Queue` ***Object Methods***	
`qsize()`	Returns queue size (approximate, since queue may be getting updated by other threads)
`empty()`	Returns `TRue` if queue empty, `False` otherwise
`full()`	Returns `true` if queue full, `False` otherwise
`put(item, block=0)`	Puts *`item`* in queue, if *`block`* given (not 0), block until room is available
`get(block=0)`	Gets *`item`* from queue, if *`block`* given (not 0), block until an item is available

Without further ado, we present the code for `prodcons.py`, shown in Example 18.9.

Example 18.9. Producer-Consumer Problem (`prodcons.py`)

We feature an implementation of the Producer-Consumer problem using Queue objects and a random number of goods produced (and consumed). The producer and consumer are individuallyand concurrentlyexecuting threads.

```
#!/usr/bin/env python

from random import randint
from time import  sleep
from Queue import  Queue
from myThread import MyThread

def writeQ(queue):
    print 'producing object for Q...',
    queue.put('xxx', 1)
    print "size now", queue.qsize()

def readQ(queue):
    val = queue.get(1)
    print 'consumed object from Q... size now', \
            queue.qsize()

```

```
def writer(queue, loops):
    for i in range(loops):
        writeQ(queue)
        sleep(randint(1, 3))

def reader(queue, loops):
    for i in range(loops):
        readQ(queue)
        sleep(randint(2, 5))

funcs = [writer, reader]
nfuncs = range(len(funcs))

def main():
    nloops = randint(2, 5)
    q = Queue(32)

    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (q, nloops),
            funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()

    print 'all DONE'

if __name__ == '__main__':
    main()
```

Here is the output from one execution of this script:

```
$ prodcons.py
starting writer at: Sun Jun 18 20:27:07 2006
producing object for Q... size now 1
starting reader at: Sun Jun 18 20:27:07 2006
consumed object from Q... size now 0
producing object for Q... size now 1
consumed object from Q... size now 0
producing object for Q... size now 1
producing object for Q... size now 2
producing object for Q... size now 3
consumed object from Q... size now 2
consumed object from Q... size now 1
writer finished at: Sun Jun 18 20:27:17 2006
consumed object from Q... size now 0
reader finished at: Sun Jun 18 20:27:25 2006
all DONE
```

As you can see, the producer and consumer do not necessarily alternate in execution. (Thank goodness for random numbers!) Seriously, though, real life is generally random and non-deterministic.

Line-by-Line Explanation

Lines 16

In this module, we will use the `Queue.Queue` object as well as our thread class `myThread.MyThread`, which we gave in Example 18.7. We will use `random.randint()` to make production and consumption somewhat varied, and also grab the usual suspects from the time module.

Lines 816

The `writeQ()` and `readQ()` functions each have a specific purpose, to place an object in the queuewe are using the string `'xxx'`, for exampleand to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

Lines 1826

The `writer()` is going to run as a single thread whose sole purpose is to produce an item for the queue, wait for a bit, then do it again, up to the specified number of times, chosen randomly per script execution. The `reader()` will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the writer sleeps is in general shorter than the amount of time the reader sleeps. This is to discourage the reader from trying to take items from an empty queue. By giving the writer a shorter time period of waiting, it is more likely that there will already be an object for the reader to consume by the time their turn rolls around again.

Lines 2829

These are just setup lines to set the total number of threads that are to be spawned and executed.

Lines 3147

Finally, we have our `main()` function, which should look quite similar to the `main()` in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single threaded program that attempts to do all of the tasks.

In this chapter, we illustrated how a single-threaded process may limit an application's performance. In particular, programs with independent, non-deterministic, and non-causal tasks that execute sequentially can be improved by division into separate tasks executed by individual threads. Not all applications may benefit from multithreading due to overhead and the fact that the Python interpreter is a single-threaded application, but now you are more cognizant of Python's threading capabilities and can use this tool to your advantage when appropriate.

PREV NEXT

18.6. Related Modules

The table below lists some of the modules you may use when programming multithreaded applications.

Table 18.6. Threading-Related Standard Library Modules

Module	*Description*
`tHRead`	Basic, lower-level thread module
`threading`	Higher-level threading and synchronization objects
`Queue`	Synchronized FIFO queue for multiple threads
`mutex`	Mutual exclusion objects
`SocketServer`	TCP and UDP managers with some threading control

18.7. Exercises

18-1. *Processes versus Threads*. What are the differences between processes and threads?

18-2. *Python Threads*. Which type of multithreaded application will tend to fare better in Python, I/O-bound or CPU-bound?

18-3. *Threads*. Do you think anything significant happens if you have multiple threads on a multiple CPU system? How do you think multiple threads run on these systems?

18-4. *Threads and Files*. Update your solution to Exercise 9-19, which obtains a byte value and a file name, displaying the number of times that byte appears in the file. Let's suppose this is a really big file. Multiple readers in a file is acceptable, so create multiple threads that count in different parts of the file so that each thread is responsible for a certain part of the file. Collate the data from each thread and provide the summed-up result. Use your `timeit()` code to time both the single threaded version and your new multithreaded version and say something about the performance improvement.

18-5. *Threads, Files, and Regular Expressions*. You have a very large mailbox fileif you don't have one, put all of your e-mail messages together into a single text file. Your job is to take the regular expressions you designed in Chapter 15 that recognize e-mail addresses and Web site URLs, and use them to convert all e-mail addresses and URLs in this large file into live links so that when the new file is saved as an `.html` (or `.htm`) file, will show up in a Web browser as live and clickable. Use threads to segregate the conversion process across the large text file and collate the results into a single new `.html` file. Test the results on your Web browser to ensure the links are indeed working.

18-6. *Threads and Networking*. Your solution to the chat service application in the previous chapter (Exercises 16-7 to 16-10) may have required you to use heavyweight threads or processes as part of your solution. Convert that to be multithreaded `code`.

18-7. **Threads and Web Programming*. The `Crawler` up ahead in Example 19.1 is a single-threaded application that downloads Web pages that would benefit from MT programming. Update `crawl.py` (you could call it `mtcrawl.py`) such that independent threads are used to download pages. Be sure to use some kind of locking mechanism to prevent conflicting access to the links queue.

18-8. *Thread Pools*. Instead of a producer thread and a consumer thread, change the code in Example 18.9, `prodcons.py`, so that you have any number of consumer threads (a *thread pool*) which can process or consume more than one item from the `Queue` at any given moment.

18-9. *Files*. Create a set of threads to count how many lines there are in a set of (presumably large) text files. You may choose the number of threads to use. Compare the performance against a single-threaded version of this code. Hint: Review Chapter 9 (Files and I/O) exercises.

18-10. Take your solution to the previous exercise and adopt it to a task of your selection, i. e., processing a set of e-mail messages, downloading Web pages, processing RSS or Atom feeds, enhancing message processing as part of a chat server, solving a puzzle, etc.

PREV NEXT