Argparse Tutorial

author: Tshepang Lekhonkhobe

This tutorial is intended to be a gentle introduction to `argparse`, the recommended command-line parsing module in the Python standard library. This was written for argparse in Python 3. A few details are different in 2.x, especially some exception messages, which were improved in 3.x.

> **Note:** There are two other modules that fulfill the same task, namely `getopt` (an equivalent for `getopt()` from the C language) and the deprecated `optparse`. Note also that `argparse` is based on `optparse`, and therefore very similar in terms of usage.

Concepts

Let's show the sort of functionality that we are going to explore in this introductory tutorial by making use of the **ls** command:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

A few concepts we can learn from the four commands:

- The **ls** command is useful when run without any options at all. It defaults to displaying the contents of the current directory.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It's named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like **cp**, whose most basic usage is `cp SRC DEST`. The first position is *what you want copied,* and the second position is *where you want it copied to*.
- Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just showing the file names. The `-l` in that case is known as an optional argument.
- That's a snippet of the help text. It's very useful in that you can come across a program you have never used before, and can figure out how it works simply by reading its help text.

The basics

Let us start with a very simple example which does (almost) nothing:

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Following is a result of running the code:

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Here is what is happening:

- Running the script without any options results in nothing displayed to stdout. Not so useful.
- The second one starts to display the usefulness of the `argparse` module. We have done almost nothing, but already we get a nice help message.
- The `--help` option, which can also be shortened to `-h`, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message, also for free.

Introducing Positional arguments

An example:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print args.echo
```

And running the code:

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

Here is what's happening:

- We've added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Calling our program now requires us to specify an option.
- The `parse_args()` method actually returns some data from the options specified, in this case, `echo`.
- The variable is some form of 'magic' that `argparse` performs for free (i.e. no need to specify which variable that value is stored in). You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don't know what it does, other than by guessing or by reading the source code. So, let's make it a bit more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print args.echo
```


And we get:

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

optional arguments:
  -h, --help  show this help message and exit
```

Now, how about doing something even more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print args.square**2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

That didn't go so well. That's because `argparse` treats the options we give it as strings, unless we tell it otherwise. So, let's tell `argparse` to treat that input as an integer:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
16
$ python prog.py four
```



```
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

That went well. The program now even helpfully quits on bad illegal input before proceeding.

Introducing Optional arguments

So far we have been playing with positional arguments. Let us have a look on how to add optional ones:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Here is what is happening:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the `if` statement.
- The help message is a bit different.
- When using the `--verbosity` option, one must also specify some value, any value.

The above example accepts arbitrary integer values for `--verbosity`, but for our simple program, only two values are actually useful, `True` or `False`. Let's modify the code accordingly:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Here is what is happening:

- The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

Short options

If you are familiar with command line usage, you will notice that I haven't yet touched on the topic of short versions of the options. It's quite simple:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
```

```
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print "verbosity turned on"
```

And here goes:

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Note that the new ability is also reflected in the help text.

Combining Positional and Optional arguments

Our program keeps growing in complexity:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print "the square of {} equals {}".format(args.square, answer)
else:
    print answer
```

And now the output:

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
```



```
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- We've brought back a positional argument, hence the complaint.
- Note that the order does not matter.

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And the output:

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

These all look good except the last one, which exposes a bug in our program. Let's fix it by restricting the values the `--verbosity` option can accept:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
```



```
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And the output:

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

optional arguments:
  -h, --help            show this help message and exit
  -v {0,1,2}, --verbosity {0,1,2}
                        increase output verbosity
```

Note that the change also reflects both in the error message as well as the help string.

Now, let's use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of `python --help`):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
```



```
else:
    print answer
```

We have introduced another action, "count", to count the number of occurrences of a specific optional arguments:

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- Yes, it's now more of a flag (similar to `action="store_true"`) in the previous version of our script. That should explain the complaint.
- It also behaves similar to "store_true" action.
- Now here's a demonstration of what the "count" action gives. You've probably seen this sort of usage before.
- And, just like the "store_true" action, if you don't specify the `-v` flag, that flag is considered to have `None` value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn't very informative on the new ability our script has acquired, but that can always be fixed by improving the documentation for our script (e.g. via the `help` keyword argument).
- That last output exposes a bug in our program.

Let's fix:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
```



```
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And this is what it gives:

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: unorderable types: NoneType() >= int()
```

- First output went well, and fixes the bug we had before. That is, we want any value >= 2 to be as verbose as possible.
- Third output not so good.

Let's fix that bug:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```


We've just introduced yet another keyword, `default`. We've set it to `0` in order to make it comparable to the other int values. Remember that by default, if an optional argument isn't specified, it gets the `None` value, and that cannot be compared to an int value (hence the `TypeError` exception).

And:

```
$ python prog.py 4
16
```

You can go quite far just with what we've learned so far, and we have only scratched the surface. The `argparse` module is very powerful, and we'll explore a bit more of it before we end this tutorial.

Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
elif args.verbosity >= 1:
    print "{}^{} == {}".format(args.x, args.y, answer)
else:
    print answer
```

Output:

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent
```

```
optional arguments:
  -h, --help        show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Notice that so far we've been using verbosity level to *change* the text that gets displayed. The following example instead uses verbosity level to display *more* text instead:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "Running '{}'".format(__file__)
if args.verbosity >= 1:
    print "{}^{} ==".format(args.x, args.y),
print answer
```

Output:

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

Conflicting options

So far, we have been working with two methods of an `argparse.ArgumentParser` instance. Let's introduce a third one, `add_mutually_exclusive_group()`. It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense: we'll introduce the `--quiet` option, which will be the opposite of the `--verbose` one:

```
import argparse

parser = argparse.ArgumentParser()
```



```
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print answer
elif args.verbose:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
else:
    print "{}^{} == {}".format(args.x, args.y, answer)
```

Our program is now simpler, and we've lost some functionality for the sake of demonstration. Anyways, here's the output:

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

That should be easy to follow. I've added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don't know:

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y
```



```
if args.quiet:
    print answer
elif args.verbose:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
else:
    print "{}^{} == {}".format(args.x, args.y, answer)
```

Note that slight difference in the usage text. Note the `[-v | -q]`, which tells us that we can either use `-v` or `-q`, but not both at the same time:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

Conclusion

The `argparse` module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.

Python Basics

Whitespace matters! Your code will not run correctly if you use improper indentation.

```
#this is a comment
```

Basic Python Logic

```
if:
  if test:
    #do stuff if test is true
  elif test 2:
    #do stuff if test2 is true
  else:
    #do stuff if both tests are false

while:
  while test:
    #keep doing stuff until
    #test is false
```

```
for:
    for x in aSequence:
      #do stuff for each member of aSequence
      #for example, each item in a list, each
      #character in a string, etc.

    for x in range(10):
      #do stuff 10 times (0 through 9)

    for x in range(5,10):
      #do stuff 5 times (5 through 9)
```

Python Strings

A string is a sequence of characters, usually used to store text.

creation: the_string = “Hello World!”
the_string = ‘Hello World!’

accessing: the_string[4] returns ‘o’
splitting: the_string.split(‘ ‘) returns [‘Hello’, ‘World!’]
the_string.split(‘r”) returns [‘Hello Wo’, ‘ld!’]

To join a list of strings together, call `join()` as a method of the string you want to separate the values in the list (‘’ if none), and pass the list as an argument. Yes, it’s weird.

```
words = [“this”, ‘is’, ‘a’, ‘list’, ‘of’, “strings”]
‘ ‘.join(words)          returns “This is a list of strings”
‘ZOOL’.join(words)       returns “ThisZOOLisZOOLaZOOLlistZOOLofZOOLstrings”
‘’.join(words)           returns “Thisisalistofstrings”
```

String Formatting: similar to `printf()` in C, uses the `%` operator to add elements of a tuple into a string

```
this_string = “there”
print “Hello %s!”%this_string returns “Hello there!”
```

Python Tuples

A tuple consists of a number of values separated by commas. They are useful for ordered pairs and returning several values from a function.

creation:
```
emptyTuple = ()
singleItemTuple = (“spam”,) <-- note the comma!
thistuple = 12, 89, ‘a’
thistuple = (12, 89, ‘a’)
```

accessing: `thistuple[0]` returns 12

Python Dictionaries

A dictionary is a set of key:value pairs. All keys in a dictionary must be unique.

creation:
```
emptyDict = {}
thisdict = {‘a’:1, ‘b’:23, ‘c’:”eggs”}
```

accessing: `thisdict[‘a’]` returns 1

deleting: `del thisdict[‘b’]`

finding:

`thisdict.has_key(‘e’)`	returns False
`thisdict.keys()`	returns [‘a’, ‘c’]
`thisdict.items()`	returns [(‘a’, 1), (‘c’, ‘eggs’)]
`‘c’ in thisdict`	returns True
`‘paradimethylaminobenzaldehyde’ in thisdict`	returns False

Python List Manipulation

One of the most important data structures in Python is the list. Lists are very flexible and have many built-in control functions.

creation:	`thelist = [5,3,‘p’,9,‘e’]`		`[5,3,‘p’,9,‘e’]`
accessing:	`thelist[0]`	returns 5	`[5,3,‘p’,9,‘e’]`
slicing:	`thelist[1:3]`	returns [3, ‘p’]	`[5,3,‘p’,9,‘e’]`
	`thelist[2:]`	returns [‘p’, 9, ‘e’]	`[5,3,‘p’,9,‘e’]`
	`thelist[:2]`	returns [5, 3]	`[5,3,‘p’,9,‘e’]`
	`thelist[2:-1]`	returns [‘p’, 9]	`[5,3,‘p’,9,‘e’]`
length:	`len(thelist)`	returns 5	`[5,3,‘p’,9,‘e’]`
sort:	`thelist.sort()`	no return value	`[3,5,9,‘e’,‘p’]`
add:	`thelist.append(37)`		`[3,5,9,‘e’,‘p’,37]`
return & remove:	`thelist.pop()`	returns 37	`[3,5,9,‘e’,‘p’]`
	`thelist.pop(1)`	returns 5	`[3,9,‘e’,‘p’]`
insert:	`thelist.insert(2, ‘z’)`		`[3,‘z’,9,‘e’,‘p’]`
remove:	`thelist.remove(‘e’)`		`[3,‘z’,9,‘p’]`
	`del thelist[0]`		`[‘z’,9,‘p’]`
concatenation:	`thelist + [0]`	returns [‘z’,9,‘p’,0]	`[‘z’,9,‘p’]`
finding:	`9 in thelist`	returns True	`[‘z’,9,‘p’]`

List Comprehension

A special expression enclosed in square brackets that returns a new list. The expression is of the form:
`[`*`expression`* `for` *`expr`* `in` *`sequence`* `if` *`condition`*`]` The condition is optional.

```
>>>[x*5 for x in range(5)]
[0, 5, 10, 15, 20]
>>>[x for x in range(5) if x%2 == 0]
[0, 2, 4]
```

Python Class and Function Definition

function:
```
def myFunc(param1, param2):
    “””By putting this initial sentence in triple quotes, you can
    access it by calling myFunc.__doc___”””
    #indented code block goes here
    spam = param1 + param2
    return spam
```

class:
```
class Eggs(ClassWeAreOptionallyInheriting):
    def __init__(self):
      ClassWeAreOptionallyInheriting.__init__(self)
      #initialization (constructor) code goes here
      self.cookingStyle = ‘scrambled’
    def anotherFunction(self, argument):
      if argument == “just contradiction”:
         return False
      else:
         return True

theseEggsInMyProgram = Eggs()
```

Files

open:
`thisfile = open(“datadirectory/file.txt”)` note: forward slash, unlike Windows! This function defaults to read-only

accessing:

`thisfile.read()`	reads entire file into one string
`thisfile.readline()`	reads one line of a file
`thisfile.readlines()`	reads entire file into a list of strings, one per line
`for eachline in thisfile:`	steps through lines in a file

Regex Accelerated Course and Cheat Sheet

For easy navigation, here are some jumping points to various sections of the page:

(direct link)

Characters

Character	Legend	Example	Sample Match
\d	Most engines: one digit from 0 to 9	file_\d\d	file_25
\d	.NET, Python 3: one Unicode digit in any script	file_\d\d	file_9৪
\w	Most engines: "word character": ASCII letter, digit or underscore	\w-\w\w\w	A-b_1
\w	.Python 3: "word character": Unicode letter, ideogram, digit, or underscore	\w-\w\w\w	字-ま_３
\w	.NET: "word character": Unicode letter, ideogram, digit, or connector	\w-\w\w\w	字-ま‿３
\s	Most engines: "whitespace character": space, tab, newline, carriage return, vertical tab	a\sb\sc	a b c
\s	.NET, Python 3, JavaScript: "whitespace character": any Unicode separator	a\sb\sc	a b c
\D	One character that is not a *digit* as defined by your engine's *\d*	\D\D\D	ABC
\W	*word character* as defined by your engine's *\w*	\W\W\W\W\W	*-+=)
\S	One character that is not a *whitespace character* as defined by your engine's *\s*	\S\S\S\S	Yoyo

(direct link)

Quantifiers

Quantifier	Legend	Example	Sample Match
+	One or more	Version \w-\w+	Version A-b1_1
{3}	Exactly three times	\D{3}	ABC
{2,4}	Two to four times	\d{2,4}	156
{3,}	Three or more times	\w{3,}	regex_tutorial
*	Zero or more times	A*B*C*	AAACC
?	Once or none	plurals?	plural

(direct link)

More Characters

Character	Legend	Example	Sample Match
.	Any character except line break	a.c	abc
.	Any character except line break	.*	whatever, man.
\.	A period (special character: needs to be escaped by a \)	a\.c	a.c
\	Escapes a special character	\.*\+\? \$\^\/\\	.*+? $^/\
\	Escapes a special character	\[\{\(\)\}\]	[{()}]

(direct link)

Logic

Logic	Legend	Example	Sample Match
|	Alternation / OR operand	22|33	33
(…)	Capturing group	A(ntlpple)	Apple (captures "pple")
\1	Contents of Group 1	r(\w)g\1x	regex
\2	Contents of Group 2	(\d\d)\+(\d\d)=\2\+\1	12+65=65+12
(?: …)	Non-capturing group	A(?:ntlpple)	Apple

(direct link)

More White-Space

Character	Legend	Example	Sample Match
\t	Tab	T\t\w{2}	T ab
\r	Carriage return character	see below	
\n	Line feed character	see below	
\r\n	Line separator on Windows	AB\r\nCD	AB CD
\N	Perl, PCRE (C, PHP, R…): one character that is not a line break	\N+	ABC
\h	Perl, PCRE (C, PHP, R…), Java: one horizontal whitespace character: tab or Unicode space separator		
\H	One character that is not a horizontal whitespace		
\v	.NET, JavaScript, Python, Ruby: vertical tab		
\v	Perl, PCRE (C, PHP, R…), Java: one vertical whitespace character: line feed, carriage return, vertical tab, form feed, paragraph or line separator		
\V	Perl, PCRE (C, PHP, R…), Java: any character that is not a vertical whitespace		
\R	Perl, PCRE (C, PHP, R…), Java: one line break (carriage return + line feed pair, and all the characters matched by \v)		

(direct link)

More Quantifiers

Quantifier	Legend	Example	Sample Match
+	The + (one or more) is "greedy"	\d+	12345
?	Makes quantifiers "lazy"	\d+?	1 in **1**2345
*	The * (zero or more) is "greedy"	A*	AAA
?	Makes quantifiers "lazy"	A*?	empty in AAA
{2,4}	Two to four times, "greedy"	\w{2,4}	abcd
?	Makes quantifiers "lazy"	\w{2,4}?	ab in **ab**cd

(direct link)

Character Classes

Character	Legend	Example	Sample Match
[…]	One of the characters in the brackets	[AEIOU]	One uppercase vowel
[…]	One of the characters in the brackets	T[ao]p	*Tap* or *Top*
-	Range indicator	[a-z]	One lowercase letter
[x-y]	One of the characters in the range from x to y	[A-Z]+	GREAT
[…]	One of the characters in the brackets	[AB1-5w-z]	One of either: A,B,1,2,3,4,5,w,x,y,z
[x-y]	One of the characters in the range from x to y	[-~]+	Characters in the printable section of the ASCII table.
[^x]	One character that is not x	[^a-z]{3}	A1!
[^x-y]	One of the characters **not** in the range from x to y	[^ -~]+	Characters that are **not** in the printable section of the ASCII table.
[\d\D]	One character that is a digit or a non-digit	[\d\D]+	Any characters, including new lines, which the regular dot doesn't match
[\x41]	Matches the character at hexadecimal position 41 in the ASCII table, i.e. A	[\x41-\x45]{3}	ABE

(direct link)

Anchors and Boundaries

Anchor	Legend	Example	Sample Match
	Start of string or start of line		

^	depending on multiline mode. (But when [^inside brackets], it means "not")	^abc .*	abc (line start)
$	End of string or end of line depending on multiline mode. Many engine-dependent subtleties.	.*? the end$	this is the end
\A	Beginning of string (all major engines except JS)	\Aabc[\d\D]*	abc (string... ...start)
\z	Very end of the string Not available in Python and JS	the end\z	this is...\n...**the end**
\Z	End of string or (except Python) before final line break Not available in JS	the end\Z	this is...\n...**the end**\n
\G	Beginning of String or End of Previous Match .NET, Java, PCRE (C, PHP, R…), Perl, Ruby		
\b	Word boundary Most engines: position where one side only is an ASCII letter, digit or underscore	Bob.*\bcat\b	Bob ate the cat
\b	Word boundary .NET, Java, Python 3, Ruby: position where one side only is a Unicode letter, digit or underscore	Bob.*\b\кошка\b	Bob ate the кошка
\B	Not a word boundary	c.*\Bcat\B.*	copycats

(direct link)

POSIX Classes

Character	Legend	Example	Sample Match
[:alpha:]	PCRE (C, PHP, R…): ASCII letters A-Z and a-z	[8[:alpha:]]+	WellDone88
[:alpha:]	Ruby 2: Unicode letter or ideogram	[[:alpha:]\d]+	кошка99
[:alnum:]	PCRE (C, PHP, R…): ASCII digits and letters A-Z and a-z	[[:alnum:]]{10}	ABCDE12345
[:alnum:]	Ruby 2: Unicode digit, letter or ideogram	[[:alnum:]]{10}	кошка90210
[:punct:]	PCRE (C, PHP, R…): ASCII punctuation mark Ruby: Unicode punctuation	[[:punct:]]+	?!.,:;
[:punct:]	mark	[[:punct:]]+	?.,: ¬\]

(direct link)

Inline Modifiers

None of these are supported in JavaScript. In Ruby, beware of (?s) and (?m).

Modifier	Legend	Example	Sample Match
(?i)	Case-insensitive mode (except JavaScript)	(?i)Monday	monDAY
(?s)	DOTALL mode (except JS and Ruby). The dot (.) matches new line characters (\r\n). Also known as "single-line mode" because the dot treats the entire input as a single line	(?s)From A.*to Z	From A to Z
(?m)	Multiline mode (except Ruby and JS) ^ and $ match at the beginning and end of every line	(?m)1\r\n^2$\r\n^3$	1 2 3
(?m)	In Ruby: the same as (?s) in other engines, i.e. DOTALL mode, i.e. dot matches line breaks	(?m)From A.*to Z	From A to Z
(?x)	Free-Spacing Mode mode (except JavaScript). Also known as comment mode or whitespace mode	(?x) # this is a # comment abc # write on multiple # lines [ ]d # spaces must be # in brackets	abc d
(?n)	.NET: named capture only	Turns all (parentheses) into non-capture groups. To capture, use named groups.	
(?d)	Java: Unix linebreaks only	The dot and the ^ and $ anchors are only affected by \n	

(direct link)

Lookarounds

Lookaround	Legend	Example	Sample Match
(?=…)	Positive lookahead	(?=\d{10})\d{5}	01234 in **01234**56789
(?<=…)	Positive lookbehind	(?<=\d)cat	cat in 1**cat**
(?!…)	Negative lookahead	(?!theatre)the\w+	theme
(?<!…)	Negative lookbehind	\w{3}(?<!mon)ster	Munster

(direct link)

Character Class Operations

Class Operation	Legend	Example	Sample Match
[…-[…]]	.NET: character class subtraction. One character that is in those on the left, but not in the subtracted class.	[a-z-[aeiou]]	Any lowercase consonant
[…-[…]]	.NET: character class subtraction.	[\p{IsArabic}-[\D]]	An Arabic character that is not a non-digit, i.e., an Arabic digit
[…&&[…]]	Java, Ruby 2+: character class intersection. One character that is both in those on the left and in the && class.	[\S&&[\D]]	An non-whitespace character that is a non-digit.
[…&&[…]]	Java, Ruby 2+: character class intersection.	[\S&&[\D]&&[^a-zA-Z]]	An non-whitespace character that a non-digit and not a letter.
[…&&[^…]]	Java, Ruby 2+: character class subtraction is obtained by intersecting a class with a negated class	[a-z&&[^aeiou]]	An English lowercase letter that is not a vowel.
[…&&[^…]]	Java, Ruby 2+: character class subtraction	[\p{InArabic}&&[^\p{L}\p{N}]]	An Arabic character that is not a letter or a number

(direct link)

Other Syntax

Syntax	Legend	Example	Sample Match
\K	Keep Out Perl, PCRE (C, PHP, R…), Python's alternate *regex* engine, Ruby 2+: drop everything that was matched so far from the overall match	prefix\K\d+	12