

Python Quick-Reference

Python has two major version in current use: 2.7 and 3. In many cases, these behave identially, but there are some differences. Try things out in the interactive interpreter, which you can start by typing `python` with no arguments.

For functions, classes, and modules, there's a built-in help system. Simply type `help(item)` for the documentation on *item*.

An important note about python: *scope is indentation-based!* A change of indentation is a change of scope, and you cannot mix spaces and tabs. You should probably ensure that your editor uses spaces for indentation in python, and be careful of reflexively using the tab key for indentation unless you're sure your editor will replace tabs with spaces. You can break up long statements on multiple lines, but only if it's unambiguous that you're in the middle of a statement (such as within parentheses or braces) or you use a line continuation character. I'm not going to tell you what the continuation character is, since it makes code ugly.

Terminal Output

`print` is a statement in python2.7, and a function in python3. You can treat it like a function in both, and it will generally do what you expect:

```
print('Hello world')
```

This will print a string to STDOUT.

Printing to STDERR, which is what you should do for debug messages, is more complex. There are two portable ways to do this:

```
import sys
sys.stderr.write('Hello world\n')
```

This uses the filehandle directly.

```
import sys
from __future__ import print_function
print('Hello world', file=sys.stderr)
```

This will ensure that the python3-style print function is present, even in python2.7.

Terminal Input

A simple way to read from STDIN is

```
import sys

while True:
    line = sys.stdin.readline()
    if '' == line:
        break
    print(line)
```

You can also use the `raw_input` or `input` functions, but this is where things get messy. In python2.7, `raw_input` reads a line from STDIN, while `input` calls `raw_input` and then evaluates the result as a python expression. In python3, `input` behaves like python2.7's `raw_input`, and there is no `raw_input` function. We can hack our way around this, though:

```
if 'raw_input' not in dir(__builtins__):
    raw_input = input
```

```
try:
    while True:
        line = raw_input()
        print(line)
except:
    pass
```

Note that both of these require some special way to handle the end of file: either testing against an empty string or handling an exception. A third way is:

```
import sys

for line in sys.stdin:
    print(line)
```

Note how much simpler this is! This is, in general, how we would read from any file.

Files

Python has an `open` function, that opens files for reading or writing, potentially in binary mode, and possibly (for writing) in append mode. See `help(open)` for details. We generally want to use these with the `with` keyword, which provides automatic file closing and other cleanup:

```
with open('input_file.txt') as in_file:
    for line in in_file:
        pass # This is a no-op

with open('output_file.txt', 'w') as out_file:
    out_file.write('Hello world!\n')

with open('output_file.txt', 'w') as out_file:
    print('Hello world!', file=out_file)
```

Scalar Types

Python has integers and floating-point numbers. Unlike many languages, all integers are arbitrary-length, as long as you have enough memory to represent them.

Python2.7 has strings, which do double-duty as byte arrays. In python3, there is a separate bytes type, and strings are utf-8 encoded by default. Most of the time, you can ignore these differences. Strings can be single-quoted or double-quoted. There isn't much reason to prefer one over the other, though if you want to use the quote character in the string, you'll have to escape it:

```
s1 = "How's it goin'?"
s2 = 'How\'s it goin\''
```

Iterable Types

Python also has lists and tuples. The difference is that a list can be modified, a tuple cannot:

```
list1 = [ 1, 2, 3, 4 ] # Initialize a list with elements
list2 = []             # Create an empty list
```

```
list2.append(1)          # Append an item to the list
list3 = list()          # Another way to create an empty list
list3.extend([1,2,3])    # Add multiple items to the list

tuple1 = ( 1, 2, 3, 4)  # Create a tuple with explicit entries
tuple2 = tuple(list1)   # Create a tuple from another iterable
```

There are other iterable types, defined by particular methods they have.

Lists and tuples can be accessed by indexes:

```
list1[0]    # first element
list1[-1]   # last element
```

They also support *slicing*:

```
list1[1:3]   # returns [ list1[1], list1[2] ]
list1[2:]    # returns [ list1[2], ..., list1[-1] ]
list1[:3]    # returns [ list1[0], list1[1], list1[2] ]
list1[0:3:2] # returns [ list1[0], list1[2] ]
list1[0::2]  # returns all even-indexed entries
list1[::2]   # same
```

You can iterate over the elements of a list or tuple:

```
for v in list1:
    print(v)
```

Note that a string (or byte string) can also be indexed like a list and iterated over.

If you want to create an iterable of integers, you can use the `range` function:

```
stop_val = 10
start_val = 1
step_val = 2
range(stop_val)           # range of ints from 0 through 9
range(start_val, stop_val) # range of ints from 1 through 9
range(start_val, stop_val, step_val) # odd ints from 1 through 9
```

Dictionaries

A python dictionary, or dict, is a map type.

```
d = {}                # Create an empty dict
d = dict()            # Create an empty dict
d = { 'a': 1, 'b': 2 } # Create a dict with initial values
```

Keys and values can be of any type, and python does not require keys or values to be uniform in type:

```
d = dict()
d[1] = 'a'
d['a'] = 2
```

Dictionaries are also iterable, though the iterator will be the keys, in some order:

```
for k in d:
    print(d[k])
```

dict has a number of useful methods (see `help(dict)` for more):

<i>Method</i>	<i>Returns</i>
<code>keys()</code>	an iterable containing the keys
<code>values()</code>	an iterable containing the values
<code>items()</code>	an iterable containing (key,value) tuples
<code>get(k)</code>	value for key k, or None if not present
<code>get(k,x)</code>	value for key k, or x if not present
<code>pop(k)</code>	value for key k, removing entry from dict

None

Python has a special type called `NoneType`, which has a single instance, named **None**. This is roughly python's equivalent of null.

List Comprehensions

Python has some functional programming elements, one of which is list comprehensions. Here's a simple example:

```
[ x**2 for x in xs ]
```

This takes a list of values named `xs` and returns a list of the values squared. These can be combined extensively:

```
[ x*y for x in xs for y in ys ]
```

The order can matter:

```
d = dict()
d['a'] = [ 1,2,3 ]
d['b'] = [ 4,5,6 ]
d['c'] = [ 7,8,9 ]
```

```
print([ x for k in d for x in d[k] ])
```

Formatted Strings

The simple way to construct a formatted string is to use the string class's `format` function:

```
s1 = 'This is {} test'.format('a')
s2 = 'The square of {} is {}'.format(2,4)
s3 = '{1} is the square of {0}'.format(2,4)
s4 = 'The first 5 powers of {x} are {pows}'.format(
    x=2,
    pows=[2**e for e in range(1,6)]
)
```

Control Flow

Like any good language, python has a number of control flow expressions. Here are a few:

```
if boolean_expression:
    do_something
elif boolean_expression_2:
```

```

        do_something_else
else:
    do_default_thing

```

Both `elif` and `else` are optional.

```

for x in xs:
    do_something

```

We've seen this before; it's a simple for loop

```

while boolean_expression:
    do_something

```

In all of these, `boolean_expression` is just something that evaluates to `True` or `False` (python's boolean constants). Python will coerce things:

- 0 is `False`
- any other number is `True`
- "" (empty string) is `False`
- any other string is `True`
- `None` is `False`
- [] (empty list) is `False`
- any other list is `True`

Combining Lists

We've already seen `list.extend()` as a way to append an entire list to another. Sometimes we want to do other things, though. Say we have a list of items, and we'd like to do something that involves their list index. We could do this:

```

for i in len(xs):
    print('item {} of xs is {}'.format(i,xs[i]))

```

There's another way we can do this, though:

```

for (i,v) in zip(range(len(xs)), xs):
    print('item {} of xs is {}'.format(i,v))

```

In this case, it doesn't seem to buy us much, but if we've read in two sequences from two different sources, but we know they should be correlated, then we could use:

```

for (a,b) in zip(a_list, b_list):
    do_something

```

The `zip` function can take multiple sequences, and will truncate the resulting tuple to the length of the shortest sequence.

Functions

Python has functions. It even has anonymous functions. Let's start with normal functions:

```

def my_func():
    pass

```

This defines a function named `my_func` that takes no arguments and does nothing, returning `None`.

```

def my_func(a):
    pass

```

Now we've added an argument to our function. Arguments can be passed based on position or name:

```
my_func(1)
my_func(a=1)
```

The latter is nice, because you don't have to worry about argument order:

```
def my_func(a,b):
    pass
my_func(b=1,a=2)
```

A common python idiom is to define very flexible functions like:

```
def my_func(a,b, *args, **kwargs):
    pass
```

This means we can provide additional positional parameters, which are then captured by `*args`, as well as named (keyword) arguments, which are captured by `**kwargs`. In this case, `args` is a tuple, and `kwargs` is a dict.

A function can return a value. If there is no return statement, the return value is `None`:

```
def my_func():
    return 'a'
```

What about anonymous functions? We define these with the `lambda` keyword:

```
f = lambda x: x**2
f(2)
```

This doesn't look like it gives us a lot of advantages over named functions, but it can be extremely handy:

```
num_output = map(lambda x: int(x,16), output)
```

```
def my_func(a,b):
    return a*b
f = lambda a: my_func(a,2)
```

Classes

Without going into a lot of detail, python has a rich type system. Here's a simple class:

```
class Foo(object):
    def __init__(self,a):
        self.a = a
```

This defines a type `Foo` and a constructor that takes two values. Here's how we create an instance:

```
foo = Foo(1)
```

By calling the class name as a function, python automatically makes this a call to the `__init__` method, with the newly allocated instance as the first argument, named `self` by convention.

Other methods can be defined similarly. Any instance method should have `self` as the first argument. Methods are otherwise almost identical to normal functions.

Python is duck-typed. That is, if it looks like a duck and acts like a duck, it's a duck. When you use a value, if it conforms to the expected interface, you're good.

You can query an object for its methods and data elements:

```
dir(foo)
foo.__dict__
```

Modules

A module is a python library. We’ve already used the `sys` and **future** modules. To use a module `foo`, you need to import it:

```
import foo
```

Now anything defined in `foo`, say a method “bar”, can be accessed through `foo`’s namespace:

```
foo.bar
```

We can also import things into the current namespace:

```
from foo import bar
from foo import *
```

The first line means we can reference “bar” without “foo.”, but the second means we can reference *everything* in `foo` without the namespace. This is generally a bad idea, because it makes it less clear where a function or class comes from. Some packages work much better with this type of import, however, like `scapy`:

```
from scapy.all import *
```

How do we create a module? We’ll keep it easy, and only consider single-file modules. Feel free to look up more complex modules. If you want to create a module named `foo`, you would simply create a file named “`foo.py`”, and define functions, classes, and variables in it as normal. Now, when you import `foo`, all of those will exist within `foo`’s namespace.

`foo.py`:

```
def bar(a):
    print('foo: {}'.format(a))
```

top-level script:

```
import foo
```

```
foo.bar(3)
```

Useful Modules

The `sys` module is the one you’re most likely to import. It has a lot of functions, but one of its most useful elements is the `sys.argv` list. This contains the positional parameters to the script, in the order provided on the command line. The first element is the script name.

```
import sys
for arg in sys.argv:
    print('We were called with argument {}'.format(arg))
```

The `random` module is also very useful; it provides random numbers:

```
import random
r = random.Random()
r.choice(['a','b','c']) # choose a random element from the list
r.sample(range(100),5)  # choose 5 unique elements from [0,100)
r.randint(5,10)         # choose an integer in the range [5,10]
r.uniform(5,10)         # choose a float in the range [5,10)
r.gauss(75,10)          # choose a gaussian-distributed float with mean
                        # 75 and standard deviation 10
```

The subprocess module lets you call other processes, potentially capturing their output. See <https://docs.python.org/2/library/subprocess.html> or <https://docs.python.org/3/library/subprocess.html>, depending on which version of python you're using.

Finally, the argparse module is a great way to process command-line arguments, if you need something fancier than just `sys.argv`. Here's an example to illustrate:

```
from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument('-s', '--students',
                    dest='students',
                    default='enrollments.json',
                    help='JSON file containing the student enrollments'
                    )
parser.add_argument('-g', '--groups',
                    dest='groups',
                    default='teams.json',
                    help='JSON file containing the student groups'
                    )
args = parser.parse_args()

students = list()
with open(args.students) as f:
    students = json.load(f)
```

See the documentation for details; there's a lot to see here.

A Complete Script

This doesn't do much useful, but it should work, when saved to a file and made executable (see the bash and filesystem quick-refs):

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

import random
import sys

count = int(sys.argv[1])
min = int(sys.argv[2])
max = int(sys.argv[3])

r = random.Random()

nums = [ r.uniform(min,max) for _ in range(count) ]

print('Generated {n} random numbers between {a} and {b}'.format(
    n=count,
    a=min,
    b=max ))
print('Values: {}'.format(nums))
```