Python tutorial

Contents

Prerequisites

This tutorial assumes you are familiar with at least one programing language (other than Python) and important programing concepts such as control flow statements (e.g. while/for loops).

The Basics

Getting Python

You can download Python from python.org. The latest version of Python is 3.4. In this tutorial, however, we use Python 2.7. Make sure to get that version if you are using this tutorial.

However, Python 3.4 is not that different from Python 2.7 (you can read about the specific differences here). If you are learning Python as a beginner (having never learned Python before), we would advise you to learn Python 3. We suggest still using this tutorial but refer to the differences as and when required. Note that Python 3 is not backward-compatible with Python 2.

If you have access to *esus*, python is already installed there.

Google Colab may be the easiest way to get up and running with Python. It is an online interactive Python environment, which includes many external libraries such as NLTK and scikit-learn without having to download and/ or install.

Running Python

In Unix you can execute a python code using the following command (assuming your code exists inside *myfirst.py* file which resides in the current directory):

```
Python myfirst.py
```

The Zen of Python

Python has a somewhat Zen description of its design principles.

One of the most discussed of these is:
There should be one—and preferably only one—obvious way to do it.

Code written in accordance with this "obvious" way (which may not be obvious at all to a newcomer) is often described as "Pythonic." We will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Whitespace Formatting

Many languages use curly braces to delimit blocks of code. Python uses indentation:

```
for i in [1, 2, 3, 4, 5]:
    print i                     # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print j                 # first line in "for j" block
        print i + j             # last line in "for j" block
    print i                     # last line in "for i" block
print "done looping"
```

This makes Python code very readable, but it also means that you have to be very careful with your formatting. Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

and for making code easier to read:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

```
two_plus_three = 2 + \
                 3
```

Modules

Certain features of Python are not loaded by default. These include both features included as part of the language as well as third-party features that you download yourself. In order to use these features, you'll need to `import` the modules that contain them.
One approach is to simply import the module itself:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Here `re` is the module containing functions and constants for working with regular expressions. After this type of `import` you can only access those functions by prefixing them with `re.`.
If you already had a different `re` in your code you could use an alias:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

You might also do this if your module has an unwieldy name or if you're going to be typing it a lot. For example, when visualizing data with `matplotlib` (the most popular plotting library for Python), a standard convention is:

```
import matplotlib.pyplot as plt
```

If you need a few specific values from a module, you can import them explicitly and use them without qualification:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

If you were a bad person, you could import the entire contents of a module into your namespace, which might inadvertently overwrite variables you've already defined:

```
match = 10
from re import *    # uh oh, re has a match function
print match         # "<function re.match>"
```

However, since you are not a bad person, you won't ever do this.

Arithmetic

Python 2.7 uses integer division by default, so that `5 / 2` equals `2`. Almost always this is not what we want, so we will always start our files with:

```
from __future__ import division
```

after which `5 / 2` equals `2.5`. In the handful of cases where we need integer division, we can get it with a double slash: `5 // 2`.

Functions

A function is a rule for taking zero or more inputs and returning a corresponding output. In Python, we typically define functions using `def`:

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Python functions are *first-class*, which means that we can assign them to variables and pass them into functions just like any other arguments:

```
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double            # refers to the previously defined function
x = apply_to_one(my_double)   # equals 2
```

Function parameters can also be given default arguments, which only need to be specified when you want a value other than the default:

```
def my_print(message="my default message"):
    print message

my_print("hello")   # prints 'hello'
my_print()          # prints 'my default message'
```

It is sometimes useful to specify arguments by name:

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5)  # returns -5
subtract(b=5)   # same as previous
```

Strings

Strings can be delimited by single or double quotation marks (but the quotes have to match):

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python uses backslashes to encode special characters. For example:

```
tab_string = "\t"       # represents the tab character
len(tab_string)         # is 1
```

If you want backslashes as backslashes (which you might in Windows directory names or in regular expressions), you can create *raw* strings using `r""`:

```
not_tab_string = r"\t"  # represents the characters '\' and 't'
len(not_tab_string)     # is 2
```

You can create multiline strings using triple-[double-]-quotes:

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

Exceptions

When something goes wrong, Python raises an *exception*. Unhandled, these will cause your program to crash. You can handle them using `try` and `except`:

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

Although in many languages exceptions are considered bad, in Python there is no shame in using them to make your code cleaner.

Lists

Probably the most fundamental data structure in Python is the `list`. A list is simply an ordered collection. (It is similar to what in other languages might be called an array, but with some added functionality.)

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)     # equals 3
list_sum    = sum(integer_list)     # equals 6
```

You can get or set the *n*th element of a list with square brackets:

```
x = range(10)   # is the list [0, 1, ..., 9]
zero = x[0]     # equals 0, lists are 0-indexed
one = x[1]      # equals 1
nine = x[-1]    # equals 9, 'Pythonic' for last element
eight = x[-2]   # equals 8, 'Pythonic' for next-to-last element
x[0] = -1       # now x is [-1, 1, 2, 3, ..., 9]
```

You can also use square brackets to “slice” lists:

```
first_three   = x[:3]               # [-1, 1, 2]
three_to_end = x[3:]                # [3, 4, ..., 9]
one_to_four = x[1:5]                # [1, 2, 3, 4]
last_three = x[-3:]                 # [7, 8, 9]
without_first_and_last = x[1:-1]    # [1, 2, ..., 8]
copy_of_x = x[:]                    # [-1, 1, 2, ..., 9]
```

Python has an `in` operator to check for list membership:

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

This check involves examining the elements of the list one at a time, which means that you probably shouldn't use it unless you know your list is pretty small (or unless you don't care how long the check takes).

It is easy to concatenate lists together:

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x is now [1,2,3,4,5,6]
```

If you don't want to modify `x` you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]       # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)       # x is now [1, 2, 3, 0]
y = x[-1]         # equals 0
z = len(x)        # equals 4
```

It is often convenient to *unpack* lists if you know how many elements they contain:

```
x, y = [1, 2]    # now x is 1, y is 2
```

although you will get a `ValueError` if you don't have the same numbers of elements on both sides.

It's common to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]    # now y == 2, didn't care about the first element
```

Tuples

Tuples are lists' immutable cousins. Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple. You specify a tuple by using parentheses (or nothing) instead of square brackets:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

Tuples are a convenient way to return multiple values from functions:

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)    # equals (5, 6)
s, p = sum_and_product(5, 10) # s is 15, p is 50
```

Tuples (and lists) can also be used for *multiple assignment*:

```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

Another fundamental data structure is a dictionary, which associates *values* with *keys* and allows you to quickly retrieve the value corresponding to a given key:

```
empty_dict = {}                        # Pythonic
empty_dict2 = dict()                   # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 }   # dictionary literal
```

You can look up the value for a key using square brackets:

```
joels_grade = grades["Joel"]           # equals 80
```

But you'll get a `KeyError` if you ask for a key that's not in the dictionary:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

You can check for the existence of a key using `in`:

```
joel_has_grade = "Joel" in grades     # True
kate_has_grade = "Kate" in grades     # False
```

Dictionaries have a `get` method that returns a default value (instead of raising an exception) when you look up a key that's not in the dictionary:

```
joels_grade = grades.get("Joel", 0)   # equals 80
kates_grade = grades.get("Kate", 0)   # equals 0
no_ones_grade = grades.get("No One")  # default default is None
```

You assign key-value pairs using the same square brackets:

```
grades["Tim"] = 99                    # replaces the old value
grades["Kate"] = 100                  # adds a third entry
num_students = len(grades)            # equals 3
```

Dictionaries are frequently used as a simple way to represent structured data:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Besides looking for specific keys we can look at all of them:

```
tweet_keys   = tweet.keys()     # list of keys
tweet_values = tweet.values()   # list of values
tweet_items  = tweet.items()    # list of (key, value) tuples

"user" in tweet_keys            # True, but uses a slow list in
"user" in tweet                 # more Pythonic, uses faster dict in
"joelgrus" in tweet_values      # True
```

Dictionary keys must be immutable; in particular, you cannot use `lists` as keys. If you need a multipart key, you should use a `tuple` or figure out a way to turn the key into a string.

Defaultdict

Imagine that you're trying to count the words in a document. An obvious approach is to create a dictionary in which the keys are words and the values are counts. As you check each word, you can increment its count if it's already in the dictionary and add it to the dictionary if it's not:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

You could also use the "forgiveness is better than permission" approach and just handle the exception from trying to look up a missing key:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

A third approach is to use `get`, which behaves gracefully for missing keys:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Every one of these is slightly unwieldy, which is why `defaultdict` is useful.

A `defaultdict` is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zero-argument function you provided when you created it. In order to use `defaultdict`s, you have to import them from `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

They can also be useful with `list` or `dict` or even your own functions:

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}
```

These will be useful when we're using dictionaries to "collect" results by some key and don't want to have to check every time to see if the key exists yet.

Counter

A `Counter` turns a sequence of values into a `defaultdict(int)`-like object mapping keys to counts. We will primarily use it to create histograms:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
```

This gives us a very simple way to solve our `word_counts` problem:

```
word_counts = Counter(document)
```

A `Counter` instance has a `most_common` method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print word, count
```

Sets

Another data structure is `set`, which represents a collection of *distinct* elements:

```
s = set()
s.add(1)       # s is now { 1 }
s.add(2)       # s is now { 1, 2 }
s.add(2)       # s is still { 1, 2 }
x = len(s)     # equals 2
```

```
y = 2 in s     # equals True
z = 3 in s     # equals False
```

We'll use sets for two main reasons. The first is that `in` is a very fast operation on sets. If we have a large collection of items that we want to use for a membership test, a set is more appropriate than a list:

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list     # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # very fast to check
```

The second reason is to find the *distinct* items in a collection:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                # 6
item_set = set(item_list)                 # {1, 2, 3}
num_distinct_items = len(item_set)        # 3
distinct_item_list = list(item_set)       # [1, 2, 3]
```

Arrays/Matrices

Although arrays and matrices can be represented as lists and lists of lists, it is recommended that you use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included. Read the NumPy tutorial to get started.

Control Flow

As in most programming languages, you can perform an action conditionally using `if`:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

You can also write a *ternary* if-then-else on one line:

```
parity = "even" if x % 2 == 0 else "odd"
```

Python has a `while` loop:

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

although more often we'll use `for` and `in`:

```
for x in range(10):
    print x, "is less than 10"
```

If you need more-complex logic, you can use `continue` and `break`:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print x
```

This will print `0`, `1`, `2`, and `4`.

Truthiness

Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False
```

Python uses the value `None` to indicate a nonexistent value. It is similar to other languages' `null`:

```
x = None
print x == None    # prints True, but is not Pythonic
print x is None    # prints True, and is Pythonic
```

Python lets you use any value where it expects a Boolean. The following are all "Falsy":

- `False`
- `None`
- `[]` (an empty `list`)
- `{}` (an empty `dict`)
- `""`
- `set()`
- `0`
- `0.0`

Pretty much anything else gets treated as `True`. This allows you to easily use `if` statements to test for empty lists or empty strings or empty dictionaries or so on. It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
```

```
if s:
    first_char = s[0]
else:
    first_char = ""
```

A simpler way of doing the same is:

```
first_char = s and s[0]
```

since `and` returns its second value when the first is "truthy," the first value when it's not. Similarly, if `x` is either a number or possibly `None`:

```
safe_x = x or 0
```

is definitely a number.

Python has an `all` function, which takes a list and returns `True` precisely when every element is truthy, and an `any` function, which returns `True` when at least one element is truthy:

```
all([True, 1, { 3 }])   # True
all([True, 1, {}])      # False, {} is falsy
any([True, 1, {}])      # True, True is truthy
all([])                 # True, no falsy elements in the list
any([])                 # False, no truthy elements in the list
```

The Not-So-Basics

Here we'll look at some more-advanced Python features that we'll find useful for working with data.

Sorting

Every Python list has a `sort` method that sorts it in place. If you don't want to mess up your list, you can use the `sorted` function, which returns a new list:

```
x = [4,1,2,3]
y = sorted(x)     # is [1,2,3,4], x is unchanged
x.sort()          # now x is [1,2,3,4]
```

By default, `sort` (and `sorted`) sort a list from smallest to largest based on naively comparing the elements to one another.

If you want elements sorted from largest to smallest, you can specify a `reverse=True` parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with `key`:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]
```

```
# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

args and kwargs

Let's say we want to create a higher-order function that takes as input some function `f` and returns a new function that for any input returns twice the value of `f`:

```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```

This works in some cases:

```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)          # 8 (== ( 3 + 1) * 2)
print g(-1)         # 0 (== (-1 + 1) * 2)
```

However, it breaks down with functions that take more than a single argument:

```
def f2(x, y):
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g() takes exactly 1 argument (2 given)
```

What we need is a way to specify a function that takes arbitrary arguments. We can do this with argument unpacking and a little bit of magic:

```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")

# prints
#  unnamed args: (1, 2)
#  keyword args: {'key2': 'word2', 'key': 'word'}
```

That is, when we define a function like this, `args` is a tuple of its unnamed arguments and `kwargs` is a `dict` of its named arguments.

stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print count
```

Reading Files

You can explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only
file_for_reading = open('reading_file.txt', 'r')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0


with open('input.txt','r') as f:
    for line in file:               # look at each line in the file
        if re.match("^#",line):     # use a regex to see if it starts with '#'
            starts_with_hash += 1   # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip()` it before doing anything with it.

Delimited Files

More frequently you'll work with files with lots of data on each line. These files are very often either *comma-separated* or *tab-separated*. Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably do). For this reason, it's pretty much always a mistake to try to parse them yourself. Instead, you should use Python's `csv` module (or the `pandas` library). For technical reasons that you should feel free to blame on Microsoft, you should always work with `csv` files in *binary* mode by including a *b* after the *r* or *w* (see Stack Overflow).

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014   AAPL    90.91

6/20/2014   MSFT    41.68

6/20/2014   FB  64.5

6/19/2014   AAPL    91.86
```

```
6/19/2014	MSFT	41.51

6/19/2014	FB	64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price

6/20/2014:AAPL:90.91

6/20/2014:MSFT:41.68

6/20/2014:FB:64.5
```

you can either skip the header row (with an initial call to `reader.next()`) or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

If you are using pandas library you can use the read_csv() function as shown below to read a delimited file.

```
import pandas as pd
df = pd.read_csv('tab_delimited_stock_prices.txt', sep = '\t')
df.head() #pretty prints the header of the file

#iterate over the rows
for index, row in df.iterrows():
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

For Further Exploration

- Book: Python for Everybody by Charles Russell Severance (freely available on web)
- Coursera's basic Python modules: e.g. https://www.coursera.org/learn/python
- There is no shortage of Python tutorials in the world. The official one is not bad.
- The official IPython tutorial is not quite as good. You might be better off with their videos and presentations. Alternatively, Wes McKinney's *Python for Data Analysis* (O'Reilly) has a really good IPython chapter.
- Book: The Hitchhiker's Guide to Python!
- Book: Learn Python the Hard Way