Cliff Notes - codeacademy.com Python Track

Sonya Sawtelle
Yale University

September 2015

This is a compilation of the majority of the material covered in codeacademy.com's Python Track. It represents the information I thought was important, or likely to forget without reviewing. It also includes some additional material for topics that codeacademy doesn't explain thoroughly (or at all). **The sections follow the module titles from the codeacademy course.** I have tried to be concise by e.g. not writing out explanations if the code example is very clear, and demonstrating multiple things in the same code example.

Contents

1 Python Syntax **2**

2 Strings and Console Output **2**

3 Conditionals and Control Flow **3**

4 Functions **5**

5 Lists and Dictionaries **8**

6 Lists and Functions **11**

7 Loops **11**

8 Advanced Topics **11**

9 Classes **14**

10 File I/O **16**

1 Python Syntax

Self-assignment shorthand can be done with most math operators. You can assign a single value to more than one variables simultaneously, and assign multiple values to multiple variables at the same time. Floor division is a useful one - it rounds the outcome down to the nearest integer. Integer division by default also rounds the answer and returns an integer. The str() method converts an object to a string object, so it can be used with the plus string operator.

```
>>> x = y = z = 1
>>> x, y, z = 10, (0,0), "abcd"

>>> num = 10.1
>>> num //= 8

>>> print "num //= 8.9 is the same as num = num//8.9 which is " + str(num)
num //= 8.9 is the same as num = num//8.9 which is 1.0

>>> print "Integer division rounds e.g. 5/2 is %i" % (5/2)
Integer division rounds e.g. 5/2 is 2
```

2 Strings and Console Output

Single or double quotes can be used for strings, with the caveat that if you use double quotes, then any double quotes within the string need to be escaped but single quotes pass fine. Vice versa for using single quotes. Single quotes show up more often in normal language, so I use double quotes by default to minimize escaping.

```
>>> print 'I don't work' + " nor do I "work""
  File "<stdin>", line 1
    print 'I don't work' + " nor do I "work""
                 ^
SyntaxError: invalid syntax
```

```
>>> print "But I don't not work" + ' and I do "work"'
But I don't not work and I do "work"

>>> print 'I also don\'t not work' + " and I also do \"work\""
I also don't not work and I also do "work"
```

Printing out variables is done with placeholders of the correct type.

```
>>> word, count, floater = "is", 1, 99.99

>>> print "%s this a %s? I am %f%% sure that this is test #%i. " \
...       "Oops, I mean %0.2f%%" % (word, "test", floater, count, floater)
is this a test? I am 99.990000% sure that this is test #1. Oops, I mean 99.99%
```

Dictionaries and local namespaces allow for more flexible string formatting. Built in locals() returns the local namespace in dictionary form.

```
>>> dic = {'name': "Ted", 'age': 23}
>>> print "Person named %(name)s is %(age)i years old" % dic
Person named Ted is 23 years old

>>> fruit, price = "apple", 2.99
>>> print "The %(fruit)s costs %(price)0.2f" % locals()
The apple costs 2.99
```

Multine quotes use triple quotations

```
''' This is a really long comment...
seriously.
really long.
'''
```

Strings behave like lists of characters. **Remember indexing starts at 0 for python.**

```
>>> print "cat"[0]
c
```

Lots of useful methods for class string. **Notice dot notation for methods specific to strings** e.g. .upper() but not for methods not specific to string class e.g. str(), len(). The continuation character \ allows you to continue code on the next line.

```
>>> word = "LaTeX"
>>> print "I can shout %s, or whisper %s. It's a short word, " \
...       "just %i letters!" % (word.upper(), word.lower(), len(word))
I can shout LATEX, or whisper latex. It's a short word, just 5 letters!

>>> "Split,this,list,with,commas".split(",")
['Split', 'this', 'list', 'with', 'commas']
```

3 Conditionals and Control Flow

Python has really nice syntax for membership and object identity logic. You can use the keywords True or False to set boolean values (rather than just 0 or 1). **Careful!** The operator **is** does not test equality of the values of two variables, but rather whether the two tags are pointing to the same object in memory! Different object types behave differently here (for complicated reasons), so be careful! Built-in function id() returns the memory address of an object.

```
>>> y = "ten"
>>> z = ["nine", "ten", "twelve"]
>>> print (y not in z), (y in z), ((y not in z) is not False)
False True False

>>> print "Notice weirdness: ", not 5, not 6.86, not "word", not 0
Notice weirdness:  False False False True

>>> a, b = 1, 1 #auto-interning for small integers
>>> print id(a), " vs ", id(b)
36332712  vs  36332712
>>> print a is b
True

>>> a, b = 100000, 100000 #no auto-interning for large integers or strings
>>> print id(a), " vs ", id(b)
48047712  vs  48047712
>>> print a is b
True

>>> a, b = "abc", "abc" #no auto-interning, but compiler is folding!
>>> print a is b
True

>>> c, d = [1,2], [1,2]
>>> print c is d
False
>>> c = d
>>> print c is d
True
```

Pretty standard operator precedence stuff.

```
**
   / * % //
        < <= > >=
                == !=
                    <>
                            is, is not
                                in, not in
                                        not
                                                and
                                                    or
```

Functions and conditional codeblocks are defined by colons and whitespace. You can pass any kind of input to a function that you want without declaring type. Return statements must be explicit.

```
>>> def myfun(nums, num):
...     if nums[0] == num:
...         print "Same"
...     elif nums[0] < num:
...         print "Less"
...     else:
...         print "Big"
...     return True
...
>>> print not myfun([1,2,3],0)
Big
False
```

4 Functions

Functions define local namespaces, **but** python distinguishes between mutable and immutable objects such that a mutable object can be modified globally from within a function using the local namespace tag for the object. This is different from how C++ treats variable tags. In Python lists, dictionaries and sets are mutable, while integers, floats, strings and tuples are immutable. For functions there is a special if/else syntax for the return statement if you want different return valuse based on conditionals - it is a bit more concise.

```
>>> a = 1 #new object (the immutable integer 1) is created
>>>         # at a memory location and the label a is affixed to it

>>> b = a #the label b is now affixed to the same object at the same location

>>> a += 1 #new object (the integer 2) is created at a different
>>>         # memory location and the label a is now associated with that object

>>> print b #the label b is still pointing at the old integer object
1


>>> c = [0, 0, 0] #new object (mutable list) is created at a memory
>>>         # location and the label c is affixed to it

>>> d = c #the label d is now affixed to the same object

>>> c[0] = 1 #the original object at the same memory location
>>>         # is modified, the tags remain associated with it

>>> print d #the label d still points to this same, but modified, list object
[1, 0, 0]
```

```
>>> def myfun(nums, num):
...     nums[0] = 1
...     num = 1
...
>>> x = [0, 0, 0] #mutable list
>>> y = 0 #immutable int
>>> myfun(x,y)
>>> print x,y
[1, 0, 0] 0

>>> def check_greater(x):
...     return True if x > 5 else False
```

Importing python packages can be done in various ways at the beginning of the document.

```
import moduleA
'''All functions/variables from moduleA are imported but all need
    to be called with dot notation (they still only exist in
    moduleA namespace)'''
moduleA.functionA()

from moduleA import functionA
'''Only functionA is imported and now functionA can be called without
    dot referencing (it exists in local namespace)'''
functionA()

from moduleA import *
'''All functions/variables from moduleA are imported and can
    be called without dot notation
    WARNING: not recommended - to hard to trace methods and refactor'''
functionA()
functionB()
```

The datetime package is very useful.

```
...
>>> from datetime import datetime
>>> now = datetime.now()
>>> print now.year, now.month, now.day
2015 10 3
```

Some useful built-in functions on different (often multiple) classes. Enumerate(sequence, start) is very powerful, it returns an enumerate object (iterable) whose next method returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over sequence. Zip(seq1, seq2...) tuples together all the sequences in element-wise fashion, stopping at the end of the shortest sequence. The star operator, *, unpacks sequences into elements - can be used in calling functions which take multiple inputs. Built-in filter(function, iterable) is equivalent to the list comprehension [item for item in iterable if function(item)]

```
>>> max([1, 2, 3.99])
3.99
>>> min([1, 2, 3.99])
1
>>> abs(-11.2)
11.2
>>> round(2.6789, 2), round(2.6789, 1)
(2.68, 2.7)
>>> sum([1, 2, 3])
6

>>> type(7), type(type("s"))
(<type 'int'>, <type 'type'>)

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]

>>> eval("1+2+3")
6

>>> l = [True, False, False]
>>> all(l), any(l)
(False, True)

>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons, start=50))
[(50, 'Spring'), (51, 'Summer'), (52, 'Fall'), (53, 'Winter')]

>>> x = [1, 2, 3]
>>> y = ["a", "b", "c", "d"]
>>> z = (-1, -2, -3, -4, -5)
>>> zip(x, y, z)
[(1, 'a', -1), (2, 'b', -2), (3, 'c', -3)]

>>> def adder(a, b):
...     return a+b
...
>>> adder(*[7,4])
11
```

```
>>> def isint(a):
...     return type(a) is int
...
>>> l = [5, "word", 1, 8.1, "c"]
>>> print filter(isint, l)
[5, 1]

>>> int('42')
42
>>> print bin(101)
0b1100101
>>> #convert the first argument to base 10 from base-second-argument
>>> int('1100101', 2)
101
```

5 Lists and Dictionaries

A list is a list of comma-separated values (items) between square brackets. Lists can contain items of different types, but usually the items all have the same type. Lists are mutable and can be nested. Lists can be sliced, and the interpretation of the first index is "beginning with index" and the interpretation of the last index is "up to but not including index". Beginning and ending indices can be omitted with the expected default of first and last index of the object. Notice some methods like .sort() are "in-place" functions, meaning they modify the variable that is calling them and return 'None'.

```
>>> animals = list("catdogfrog")

>>> print animals[:3], animals[3:6], animals [6:]
['c', 'a', 't'] ['d', 'o', 'g'] ['f', 'r', 'o', 'g']

>>> print animals[::-1]
['g', 'o', 'r', 'f', 'g', 'o', 'd', 't', 'a', 'c']


>>> test = ["t", "e", "s", "t"]

>>> test.append("y") #testy
>>> test.insert(2, "y") #teysty
>>> test.sort() #esttyy
>>> test.remove("e") #sttyy
>>> test.index("t") #1 because it is first instance of "t"
1
>>> # Treat a list like a stack
>>> n = ["firstelem", "secondelem", "thirdelem"]
>>> index = 1
>>> elem = n.pop(index)
>>> print "Popped " + str(elem) + " so now stack is " + str(n)
Popped secondelem so now stack is ['firstelem', 'thirdelem']
```

A dictionary is a mutable unordered set of **key: value** pairs, with the requirement that the keys are unique (within one dictionary) and keys are made of an immutable type. Some useful methods on dictionaries, and you can use dict comprehension syntax.

```
>>> ages = {"Bill": 62, "Dianna": 13, "Tom": 17}

>>> del ages["Bill"]
>>> ages["Dianna"] += 3
>>> ages["Betsy"] = 31 #add new entries at will! mutability!

>>> print ages
{'Betsy': 31, 'Dianna': 16, 'Tom': 17}

>>> print ages.keys(), ages.values(), ages.items()
['Betsy', 'Dianna', 'Tom'] [31, 16, 17] [('Betsy', 31), ('Dianna', 16), ('Tom', 17)]
```

A set is a mutable unordered collection with no duplicate elements. You can use set comprehension syntax.

```
>>> basket = ['apple', 'apple', 'apple', 'pear']
>>> fruit = set(basket) #removes duplicates, same as fruit = {'apple', 'pear'}
>>> fruit
set(['pear', 'apple'])

>>> print "Very quickly check if 'orange' is in 'fruit': ", 'orange' in fruit
Very quickly check if 'orange' is in 'fruit':  False

>>> a, b = set("wait"), set("weight")

>>> print "Very quickly perform set ops like the difference a - b : ", a-b
Very quickly perform set ops like the difference a - b :  set(['a'])
```

Tuples are immutable, and usually contain an heterogeneous sequence of elements that are accessed via unpacking or indexing.

```
>>> mytup = 1, 13.3, 'hello!', [0, 1, 2]

>>> myint, myfloat, myword, mylist = mytup

>>> print "Tuples can be packed and then " \
...       "unpacked - %i - %f - %s - %s" % (myint, myfloat, myword, str(mylist))
Tuples can be packed and then unpacked - 1 - 13.300000 - hello! - [0, 1, 2]
```

A list keeps order, dict and set don't (looping through dict happens in random order). A dict associates with each key a value, while list and set just contain values. A set (dict) requires items (keys) to be hashable, list doesn't. Set forbids duplicates, list and dictionary do not. Checking for membership of a value in a set (or dict, for keys) is very fast, while in a list it takes time proportional to the list's length.

Note on hashes from wikipedia: A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. One use is a data structure called a hash table, widely used in computer software for rapid data lookup. Hash functions are primarily used in hash tables, to quickly locate a data record (e.g., a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets. Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indexes), and so it will map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a bucket, and hash values are also called bucket indices. Thus, the hash function only hints at the record's location it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

List, dict and set comprehension is a comapct syntax that returns a list, dict or set respectively. A list comprehension consists of **brackets** containing an **expression** followed by a **for** clause, then zero or more **for** or **if** clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

```
>>> zoo = {'Puffin' : 104, 'Sloth' : 105, 'Python' : 106}

>>> # Missing brackets around the list comprehension
>>> print zoo[x] for x in ["Sloth", "Python"]
  File "<stdin>", line 1
    print zoo[x] for x in ["Sloth", "Python"]
                   ^
SyntaxError: invalid syntax

>>> # Now with correct bracketing
>>> print [zoo[x] for x in ["Sloth", "Python"]]
[105, 106]

>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

6 Lists and Functions

Nothing to see here...basically their way of showing that lists are mutable objects without having to actually explain it.

7 Loops

For and While loops can both include Else statements which executes when the loop exits **unless** the loop exits via a **break** statement. A while loop can be forced to execute at least once by setting the condition to True by definition and then using break as the exit condition. For loops work with lots of iterable types.

```
flag = True
while flag:
    ans = raw_input("Y or N?")
    flag = ans == "Y"
    if ans not in ["Y","N"]:
        print "I don't understand, I quit!"
        break
else:
    "Okay, NO!"
```

```
>>> for letters in "word":
...     print letters,
...
w o r d

>>> dic = {"key1": 1, "key2": 2}
>>> for keys in dic:
...     print dic[keys],
...
2 1
```

8 Advanced Topics

List comprehension is an awesome way to build lists (or sets, dicts). A list comprehension consists of brackets containing an **expression** followed by a **for** clause, then zero or more **for** or **if** clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. List comprehensions can be nested. List comprehensions can contain **if/else** logic.

```
>>> [x**2 for x in range(10) if x%2==0]
[0, 4, 16, 36, 64]

>>> [(x, x**2, x**3) for x in range(4)] #create tuples (need parentheses!)
```

```
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27)]

>>> words = ['apple', 'bravo', 'cap'] #apply methods
>>> [word.upper() for word in words]
['APPLE', 'BRAVO', 'CAP']

>>> vec = [[1,2,3], [4,5,6], [7,8,9]] #with nested for loops
>>> [num for subset in vec for num in subset]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> matrix = [
...    [1, 0, 0],
...    [2, 0, 0],
...    [3, 0, 0]]

>>> trnspd = [[row[i] for row in matrix] for i in range(3)] #with nested comps
>>> for row in trnspd:
...     print row
...
[1, 2, 3]
[0, 0, 0]
[0, 0, 0]

>>> #if, else logic in the "expression" term
>>> [x+9 if (x < 3) else x-9 if (3 <= x < 6) else x for x in range(10) if x < 8]
[9, 10, 11, -6, -5, -4, 6, 7]
```

Python permits functional programming so you can pass functions around like variables. Lambda functions are useful when you need some quick work done but won't be needing to call the function again and again. Lambda functions can consist of only a single expression, which is returned by default. Lambda functions can be defined inline basically anywhere, they are often used with certain built-ins. The built-in *reduce*(function(x,y), iterable) takes a function that must have two arguments and applies it cumulatively to the items of iterable so as to reduce the iterable to a single value. In reduce() the left argument of the function, x, is the cumulative value and the right argument, y, is the next value from the iterable.

```
# make instances of functions
def make_incrementor(n):
    return lambda x: x + n #make_incrementor(n) returns a f'n of x!

f = make_incrementor(1) #f will be a f'n tha increments its argument by 1
g = make_incrementor(10)#g will be a f'n tha increments its argument by 10

make_incrementor(1)(20) #can nest the f'n inputs! This will behave like f(20)
```

```
>>> l = [0, 1, 2, 3, 4]
>>> print filter(lambda x: x%2 == 0, l) #filter list by divisibility by 2
[0, 2, 4]

>>> print map(lambda x: x + 1, l) #add one to each elemen of the list
[1, 2, 3, 4, 5]

>>> print reduce(lambda x, y: x + y, l) #sum all the elements of the list
10
```

Built-in int() can convert strings to int, it can also convert any number represented in any base to the equivalent representation in base 10! The syntax 0b starts a binary number. Built-in bin() converts a number, binary or otherwise, to a string equivalent of the binary representation of that number. You can use 'bit masks' to selectively check the status or change the status of a specific bit(s) in a binary by bitwise and, or, xor.

```
>>> #convert the first argument to base 10 from base-second-argument
>>> print int('42'), int('1100101', 2)
42 101

>>> print bin(101), type(bin(101)), bin(0b1100)
0b1100101 <type 'str'> 0b1100
>>> print oct(101), hex(101)
0145 0x65

>>> print bin(0b1100 >> 2), bin(0b1100 << 2)
0b11 0b110000

>>> #bitwise AND comparison: '1' if both bits are on
>>> print bin(0b100101 & 0b010111)
0b101

>>> #bitwise OR comparison: '1' if either bit is on
>>> print bin(0b100101 | 0b010111)
0b110111

>>> #bitwise XOR comparison: '1' only if the bits are different!
>>> print bin(0b100101 ^ 0b010111)
0b110010

>>> print bin(0b1111 ^ 0b1100) #XORing with all one's flips the bits!
0b11

>>> print ~3, ~(-8)     #uhhhh...add one and then multiply by -1.....
-4 7

>>> print bin(0b1 <<3) #make a mask to flip the 4th bit from the right
```

```
0b1000
>>> print bin(0b10101 ^ (0b1 <<3))
0b11101
```

9 Classes

Every class definition should define the method __init__(self, optional arg1...)(). Note that *self* is passed as the first argument in all member method definitions, but not when actually accessing a method with an instance. Access attributes of an instance with dot notation. The __repr__() method tells the interpreter how to represent your class when e.g. printing. With classes you can have variables (and methods) that are available globally, only available to members of a certain class (class variables), and variables that are only available to particular instances of a class (instance variables). Instance variables and methods are referenced in the class definition by self.var notation and are inantiated as different memory objects for each instance. Class variables are don't use dot notation are created at class definition time when the interpreter encounters the class statement, so each new instance will point to the same class variable objects. This object can be changed (effective for all instances) dynamically by referencing *Class.classvar*. **Note**, a class variable that is a mutable type can also be accessed and altered by any instance, and all other instances will remain pointing to the now-altered object.

```
>>> class A:
...     #member variables
...     foo = []
...     fooimmut = 1
...     footupimm= 1, 13.3, 'hello!', [0, 1, 2]
...     def __init__(self):
...         #instance variables
...         self.bar = []
...         self.barimmut = 1
...         self.bartupimm = 1, 13.3, 'hello!', [0, 1, 2]
...
>>> a, b = A(), A()

>>> print id(a.foo), id(b.foo) #instantiated to point to same object
51161096 51161096
>>> print id(a.fooimmut), id(b.fooimmut) #instantiated to point to same object
36332712 36332712
>>> print id(a.footupimm), id(b.footupimm) #instantiated to point to same object
48946136 48946136

>>> print id(a.bar), id(b.bar) #instantiated as different objects
51160392 48977800
>>> print id(a.barimmut), id(b.barimmut) #same object b/c of interpreter cleanup
36332712 36332712
>>> print id(a.bartupimm), id(b.bartupimm) #instantiated as different objects
```

```
51084248 51082008

>>> a.foo.append([1])
>>> print id(a.foo), id(b.foo) #IDs unchanged
51161096 51161096

>>> a.fooimmut = 2
>>> print id(a.fooimmut), id(b.fooimmut) #new ID for instance 'a'
36332688 36332712

>>> c = A()
>>> print id(c.fooimmut) #will point to original object from class def'n time
36332712

>>> A.fooimmut = 7 #modify the class-level variable
>>> #all pointers still assoc'd with original object are moved over to new object
>>> print id(a.fooimmut), id(b.fooimmut)
36332688 36332568
```

Inherit from a base class by taking it as an argument of the new class definition. Overwrite inherited methods as desired. You can still call any overwritten methods of the parent class with super().

```
>>> class Employee(object):
...     def calculate_wage(self, hours):
...         self.hours = hours
...         return hours * 20.00
...
>>> class PTEmployee(Employee):
...     def calculate_wage(self, hours):
...         self.hours = hours
...         return hours * 10.00
...     def FT_wage(self, hours):
...         return super(PTEmployee, self).calculate_wage(hours)
...     def __repr__(self):
...         return "a worker with %0.2f hours under their belt" % self.hours
...
>>> milton = PTEmployee()
>>> print milton.calculate_wage(10)
100.0
>>> print milton.FT_wage(10)
200.0
>>> print milton
a worker with 10.00 hours under their belt
```

10 File I/O

Using open with the ‘w’ mode means open the file to write only to it, ‘r’ means read only and ‘r+’ mode read and write. The write() method of file objects takes only string arguments. Always call the close() method on the file object when done. The readline() method returns the next line in the file (including the newline character at the end) each time it is called. The term ‘flush the buffer’ means actually write all the stuff it was holding in buffer, it doesn’t do that until forced by e.g. close(). The *closed* attribute of a file object will return a boolean indicating the status (open or closed). The syntax **with open(”file”, ”mode”) as variable:** will automatically do open() and close() when it has finished executing the sub-block of instructions.

```
read_file = open("text.txt", "r")
# Use a second file handler to open the file for writing
write_file = open("text.txt", "w")
# Write to the BUFFER
write_file.write("This will be the first line in the file.")
print read_file.read() #NOTHING WILL BE HERE! the buffer has not been flushed
```

```
with open("text.txt", "r") as textfile:
    currentline = textfile.readline()
```