Kevin Chen

1/8/16

Python Notes

Python Quick Start

* Getting started with “Hello World”
  + print(“Hello World”)
  + print adds a new line character at the end (similar to println() in Java)
  + In mostly Unix-based OS environments, you will see a line called the “shebang line”. This line begins with a hash and an exclamation mark followed by the path to the interpreter that will be used to run the script. Example: #!/usr/bin/python3
  + In Unix-based OS, make sure your file has executable permissions by typing ls –l *filename* and looking at the file permissions. If it doesn’t have executable permissions, type chmod 755 *filename* to give it executable permissions. Then run it by ./*filename*.
* Selecting code with conditionals
  + (Multiple assignments can be done at once. Example: a, b = 0, 1 will assign 0 to a and 1 to b. Don’t declare types in front of variables.)
  + Spacing is very important in Python. Blocks are called suites in Python. Indentation is usually 4 spaces (this is the standard).
  + if *condition1*:

#Code to execute

elif *condition2*:

#Code to execute

else:

#Code to execute

* + The conditional expression “*value1* if *condition1* else *value2*” will return *value1* if *condition1* is met; otherwise, it will return *value2*.
* Repeating code with a loop
  + while loop:

while *condition*:

#Code to execute

* + For loops work with iterators, which return the next value each time you call it.

for *item* in *iterator*

#Do something with *item*

* + range(*m, n*) creates an iterator from the number *m* (inclusive) to *n* (exclusive). If first parameter is excluded, then it starts from 0.
  + (Open a text file with the open() method, passing in the name of the file. Assign it to a variable. Use the instance method readLines() of the file variable to create an iterator for the file.)
* Reusing code with a function
  + def *functionname*(parameters):

# function code

* Creating sequences with generator functions
  + Generator functions create an iterator
  + Instead of having a return statement at the end of a function, a generator function has a yield statement. The next time the function is called, it continues execution after the yield statement.
  + Often use a while-true loop, and surround the yield statement to return the next value of the iterator each time the loop runs.
* Reusing code and data with a class
  + class *classname*(*class\_to\_inherit*):

def \_\_init\_\_(*parameters*) # Constructor method

# Constructor code

#...methods…

* + Don’t need to list fields. Calling self.*fieldname* will create a field in the constructor.
  + Add *class\_to\_inherit* if this class extends some other class. Otherwise omit (*class\_to\_inherit*)
  + The first parameter of each method should be a reference to the instantiated object (and hence is often named self).
  + Instantiating: *variable\_name* = *classname*(*constructor\_parameters*)
  + When calling an object’s method, don’t pass in anything for the first parameter, self. Thus, you should be passing in one less parameter than the number of parameters specified in the object’s method header.
  + Add a return statement to return a value. (Don’t add return type in method header.)
* Greater reusability with inheritance and polymorphism
  + (Adding an underscore in front of a method name indicates that it should not be called outside of the object; that is, it should be private.)
  + One strategy to use polymorphism is to define a field across all subclasses (but have the actual value of it be different for each class), and the superclass works on the field.
* Handling errors with exceptions
  + try-catch block in Java is try-except.
  + try:

# Some code

except *TypeOfError(e.g. IOError)* as *ErrorVariable(e.g. e)*

# Do something with *ErrorVariable*

Setting up Python

* Installing Python 3 and Eclipse for Windows
  + Install Python3 and Eclipse and PyDev. (Full instructions can be found online.)
  + Python comes with a program called the idle, which is the graphical interface of the Python shell.

General Syntax

* Creating a main script
  + The first line, the shebang line, contains a pound (the comment indicator) and an exclamation point, followed by a path to the Python Interpreter. This is used for environments that run the script from a shell (e.g. UNIX-based environments). Ensure that the path is correct.
  + The last line is often if \_\_name\_\_ == “\_\_main\_\_”: main(). This line allows us to use functions before they are defined. Normally, you can just have main() instead of having the whole if statement, but the first part will only allow the line to run if the file is the main module. It won’t run when the module is included in other modules.
* Understanding whitespace in Python
  + Whitespace is very significant in Python! This is because it doesn’t use curly braces to indicate ends of a suite (or a block of code).
  + All whitespace must be consistent. Typically use four spaces before each line of code within a suite.
  + If there is just one line of code within a control structure (e.g. method), then it can be on the same line as the control structure. Example:

def main(): print(“Hello world”)

* Commenting code
  + Pound signs are comments. Everything starting from the symbol until the end of the line is ignored from the interpreter.
  + But cannot comment out code.
* Assigning values
  + The equal symbol is the assignment operator.
  + Example: a = 1. Don’t declare type in front of a.
  + type(*variable*) gets the type of *variable*.
  + Multiple assignment at once: a, b = 0, 1
  + A tuple: a = (1, 2, 3, 4, 5) An aggregate type: a = [1, 2, 3, 4, 5]
* Selecting code and values with conditionals
  + (Look above for conditional execution: if-statement syntax.)
  + (Look above for conditional expressions)
* Creating and using functions
  + (print() function takes in optional second parameter to replace the traditional new line character at the end. Syntax is: print(*string*, end=‘*endingchar*’)
  + Syntax:

def *functionname*(*parameters*):

# Function body

* + Similar with C++, you can assign a value within the parameters if you want a parameter to have a default value if the value is not passed into the function.
* Creating and using objects
  + Everything in Python is an object
  + (Look at above section: Reusing code and data with a class)

Variables, Objects, and Values

* Understanding variables and objects in Python
  + Everything in Python is an object (e.g. variables, functions, and even code).
  + Every object has an ID, Type, and Value. The ID uniquely identifies a particular instance of an object and cannot change for the life of the object. The type identifies the class of an object and cannot change for the life of the object. The value is the contents of the object. Mutable objects can change value, but immutable objects cannot.
  + In the python shell, you can initialize a variable as you would normally do. Display the contents of the variable by just entering the variable name. id(*variablename*) returns the numerical id of this object. type(x) returns the class of this object.
  + All variables in Python are first class objects. That means that what looks like a simple variable may be something more complex.
* Distinguishing mutable and immutable objects
  + Most fundamental types in Python are immutable. Examples: numbers, strings, tuples. It may seem that you are changing the value of an immutable object if you assign the variable to another value, but in reality, you are just changing the reference of the variable to refer to another object. The distinction becomes apparent when you find the object’s id using the id() function.
  + Mutable objects include: lists, dictionaries, and many other objects.
* Using numbers
  + Integers vs. floating point numbers
  + 42 / 9 returns a floating point number. 42 // 9 does integer division.
  + round(*double*) rounds a number. Can pass in a second parameter to specify how many digits you want to round to.
  + % is the remainder (modulo)
  + Casting is done as so: *type*(*value*). Example: int(42.9) converts the floating point number 42.9 to the integer 42. We are using object constructors of the *type*’s class.
* Using strings
  + Strings can be surrounded by single or double quotes.
  + Escape characters like \n
  + Raw string: want the string to be interpreted literally as it is (e.g. escape characters are interpreted specially). Do it by adding an “r” in front of the string. Used often with regular expressions.
  + Call the format() instance method of the string class to replace the curly braces within the string with the object passed in (Python 3 way). Example:

n = 42

s = ‘The number is {}’.format(n) # will display The number is 42

* + Python 2 way to do the above thing: s = ‘The number is %s’ % n
  + Use triple quotes (either single or double) easily allows you to have a string that can span multiple lines. Whenever there is a line break in the code, there is a line break in the actual string. (Putting a backslash at the end of a line is an escape character to get rid of the new line.) Often used for docstrings in functions.
* Aggregating values with lists and tuples
  + Tuples: List values separated by commas and surround all the values with parenthesis. Example: x = (1, 2, 3). Tuples are immutable (and thus faster).
  + Lists: surround values using square brackets instead of parenthesis. Example: x = [1, 2, 3]. This is mutable. Use append(*value*), insert(*value*, *index*), etc. to change it.
  + Access elements like you would in Java for arrays. Example: x = (1, 2, 3), then x[1] returns 2. Can do x[*beginIndex*:*endIndex*] to get all values from *beginIndex* (inclusive) to *endIndex* (exclusive) (called a slice).
  + Strings can be used just like tuples. Each character is an element.
  + Can use lists and tuples as an iterator.
* Creating associate lists with dictionaries
  + Basically an associative array (or hash).
  + *listname* = {*key1*: *value1*, … , *keyn*: *valuen*}
  + Using keyword argument instead:

*listname* = dict(

*key1* = *value1*,…, *keyn* = *valuen* # Don’t need quotes around each key

# if the key is a string.

­)

* + Accessing an element with a particular key: *listname*[*key*]
  + Can iterate through the elements: for *key* in *listname*:
  + Note that keys are hashed, so you can’t directly iterate in any particular order. Can sort the keys using the method sorted(*listname*)
  + Dictionaries are mutable objects. Can add values as so: *listname*[*newkey*] = *newvalue*.
* Finding the type and identity of a variable
  + In the Python Shell, id(*object*) gives you the variable of the object.
  + type(*object*) returns the type of the object.
  + The id of identical objects are the same. Exception is mutable class.
  + x == y checks to see if the values are the same, whereas x is y compares the IDs to see if they refer to exactly the same object.
* Specifying logical values with True or False
  + True and False are immutable Boolean objects. (Note: these are capitalized)

Conditionals

* Selecting code with if and else conditional statements
  + (Refer to section: “Selecting code with conditionals”)
* Setting multiple choices with elif
  + (Refer to section: “Selecting code with conditionals”)
* Understanding other strategies for multiple choices
  + Python doesn’t have switch or case control structures.
  + Normally, if you try to obtain a value from a dictionary that doesn’t have the given key, then you get an error. But instead can call the instance method get(*key*, *defaultvalue*), where *defaultvalue* is displayed if the *key* is not found in the dictionary.
* Using the conditional expression
  + (Refer to the section “Selecting code with conditionals”)

Loops

* Creating loops with while
  + (Refer to the section “Repeating code with a loop”)
* Iterating with for
  + (Refer to the section “Repeating code with a loop”)
  + All the container types (e.g. lists and strings) are iterators.
  + One of the most common types of loops in Python.
* Enumerating iterators
  + Notice that for loops doesn’t have an index. The enumerate() function solves this.
  + for *index*, *item* in enumerate(*iterator*)

# Do something with the *index* and *item*.

* Controlling loop flow with break, continue, and else
  + These controls work for both for and while loop.
  + break and continue work just like it does in Java.
  + The else block is run one time once the for loop terminates. Example:

for c in s:

#Do something

else

#Do something after the above loop terminates

Operators

* Performing simple arithmetic
  + +, -, \*, / (does not do integer division, so 5 / 3 returns 1.666667; always returns a floating point number), // (integer division), %
  + divmod(*a*, *b*) returns: (*a // b*, *a % b*), which is a tuple
  + +=, -=, \*=, /=, //=,
* Operating on bitwise values
  + (Binary: 0b*value*. Hexadecimal: 0x*value*)
  + *a* | *b* is the or-operator, *a* & *b* is the and operator, *a* ^ *b* is the ex-or operator, *a* << *b* shifts *a* by *b* bits left, *a* >> *b* shifts *a* by *b* bits right, ~*a* is the unary (one’s complement) operator.
  + Note that numbers have a sign bit, so using the unary operator on a positive number would result in a negative number, and vice versa.
* Comparing values
  + <, >, <=, >=, ==, !=
  + *x* is *y* returns if the id of *x* is the same as the id of *y*. *x* is not *y* does the opposite.
  + (Remember that the ids of immutable objects, such as integers, are the same if the objects are identical. The ids of identical mutable objects would not the same.)
* Operating on Boolean values
  + Boolean values in Python are the special True and False objects of type bool.
  + Instead of && or ||, you use the fully typed out words: “and” or “or”
* Operating on parts of a container with the slice operator
  + *listname*[*a*] returns a slice containing the index at *a*.
  + *listname*[*a*:*b*] returns a slice of the array from index *a* (inclusive) to index *b* (exclusive).
  + *listname*[*a*:*b*:*c*] returns every *c*’th element from index *a* (inclusive) to index *b* (exclusive).
  + (*listname*[:] = list(range(*parameters*)) appears to convert a range to a list, but I’m having problems trying to get this to work.
  + Can assign slices by doing: *slice* = (*values, separated, by, commas*)
* Understanding operator precedence
  + Refer to an operator precedence chart.

Regular Expressions

* Using the re-module
  + Regular expressions are a very powerful method for matching patterns in text.
  + Regular expressions are a small language in itself.
  + Implemented in Python with the “re” module
  + Basic code structure:

import re #This is not usually needed.

pattern = re.compile(*RegexPattern*)

if re.search(pattern, *string*):

#If this is running, *string* contains has the pattern.

* Searching with regular expressions
  + Use the re.search(*RegexPattern*, *string*) to do a regex search.
  + re.search returns the portion of the string that was matched with the regular expression pattern. Can group them together (calling the group() instance method and printing it out)
* Replacing with regular expressions
  + Use the function: re.sub(*RegexPattern*, *ReplaceWith*, *ReplaceFrom*). This functions replaces the text that satisfies *RegexPattern* within the string *ReplaceFrom* with the string *ReplaceWith*.
  + This function returns the replaced string (even if no replacements occur).
  + Alternatively can store the result of all matches from re.search(*RegexPattern*, *string*). Then call *string*.replace(*matches*.group(), *ReplaceWith*). This essentially separates the previous method of replacing values in two steps.
* Reusing regular expressions with re.compile
  + Every time you call re.search(*RegexPattern*, *string*), you cause Python to have to recompile the *RegexPattern* every time. Can make this more efficient with re.compile
  + Store the pattern returned by re.compile(*RegexPattern*) and then use the stored pattern whenever you need to use the regular expression pattern.
  + This also allows us the ability to use some of the regular expression module’s other features. Example: re.compile(*RegexPattern*, re.IGNORECASE) will cause the regular expression module to ignore cases within the *RegexPattern*.
  + Can substitute text with re.compile: *pattern*.sub(*ReplaceWith*, *ReplaceFrom*).

Exceptions

* Learning how exceptions work
  + Uncaught exceptions are the ones where you run a script and Python outputs a stack trace and an error message.
  + Can catch exceptions using try-except.
  + Example (IO error for opening files):

try:

fh=open(‘filename’)

except IOError as e:

#do something with e, the error message, such as printing it.

else:

#file opened successfully. Run normally.

* + You may raise your own exceptions with “raise”
* Handling exceptions
  + (*string*.strip() well strip any trailing new lines at the end of the string.)
  + Don’t specify type of error after except (so you just type “except:”) if you want to catch all types of exceptions. You can get rid of “as e” if you don’t need the error message.
  + If you have more than one line of code in the try block, execution will stop at that line in the try block if an exception occurs. So you can move anything in the else clause up to the end of the try block.
  + Can test for more than one type of exception after a try block.
* Raising exceptions
  + (Can check if a string ends with a particular text by saying calling the method: *string*.endswith(‘*stringending*’))
  + Python equivalent of throwing exceptions in Java.
  + If a particular condition occurs, call “raise *TypeOfError*(‘*string description*’)”
  + Can then handle your raised exception.

Functions

* Defining functions
  + Syntax:

def *nameOfFunction*(parameters, separated, by, commas):

#lines of code.

* + Functions must have a body. If you don’t want it to have a body yet, put “pass” inside the function.
  + Optional arguments done by giving them default values, just like in C++, by assigning them in the method signature. Example: def increment(number, incrementBy = 1):
  + Assign optional arguments to the special value, None, if you really want it to be optional.
* Using list of arguments
  + List of arguments can be used if you want the caller to pass in an arbitrary number of arguments. Do this by putting an asterisk in front of the parameter you want an arbitrary number of. Example: def func(a, b, \*list). You can then pass in 3, 4, or 100 arguments into func. The first argument will be a, the second b, and the rest list.
  + Using the list of arguments directly will result in a tuple. Can use it as an iterator.
* Using named function arguments
  + Can have function arguments be named by the caller of the function as opposed to the receiver.
  + These arguments are passed as a dictionary of key-value pairs.
  + Receiving function uses double asterisks to indicate it is receiving a dictionary. Example: def testfunc(\*\*kwargs). (kwargs is a common name used to mean key-word arguments). Use bracket notation to access the corresponding value of a key.
  + Caller of function uses the syntax *key* = *value* for each dictionary entry. Example: testfunc(one = 1, two = 2, four = 42);
  + Full Example:

def testfunc(this, that, other, \*args, \*\*kwargs):

print(‘This is a test function’, this, that, other, args, kwargs[‘one’], kwargs[‘two’], kwargs[‘four’])

def main():

testfunc(5, 6, 7, 8, 9, 10, one = 1, two = 2, four = 42)

#5 passed in as this, 6 passed in as that, 7 passed in as other

#8, 9, and 10 passed into \*args. one = 1, two = 2, four = 42 passed into \*\*kwargs

#Output: “This is a test function 5 6 7 (8, 9, 10) 1 2 42”

* + Can use the dictionary as an iterator. It iterates through the keys.
  + Tuple arguments are iterated in original order (think of List). Dictionaries are not iterated in any particular order (think of HashMap)
* Returning values from functions
  + return something at the end of a function. Example:

def testfunc():

return range(25)

* Creating a sequence with a generator function
  + (Refer to section “Creating sequences with generator functions”)
  + Example (prints 0 – 25, inclusive by creating a custom iterator):

def main():

for i in inclusive\_range(0, 25, 1):

print(i, end = ‘ ’)

def inclusive\_range(start, stop, step):

i = start

while i <= stop:

yield i

i += step

* + (Quickly assigning variables to tuple values example)

# start and stop are variables created and initialized to the two values inside the

# tuple args

(start, stop) = args

**Classes**

* Understanding classes and objects
  + (Refer to section “Reusing code and data with a class”)
  + Example:

class Duck:

def \_\_init\_\_(self, value): # Constructor

print(‘constructor’)

#Creates a local variable \_v that is an attribute of the object.

self.\_v = value

def quack(self):

print(‘Quaaack!’, self.\_v)

def walk(self):

print(‘Walks like a duck.’, self.\_v)

def main():

#Creates the donald object and prints ‘constructor’

donald = Duck(47)

print(donald.\_v) # Prints 47

donald.quack() # Prints ‘Quaaack! 47’

donald.walk() # Prints ‘Walks like a duck. 47’

donald.\_v = 40

print(donald.\_v) # Prints 40

* Using methods (Notes are combined with section “Understanding classes and objects”)
* Using object data
  + Naming convention (not required, but can be good form): Put underscore in front of variable name (e.g. \_color) in order to show that it is an attribute you will be using locally for the object. (Not used directly outside the object.)
  + Use a dictionary (key-word arguments) makes it easier to scale by having many variables. Example:

class Duck:

def \_\_init\_\_(self, \*\*kwargs):

self.variables = kwargs

def set\_varaible(self, k, v):

self\_variables[k] = v

def get\_varaible(self, k):

return self.varaiables.get(k, None)

* Understanding inheritance
  + Add subclass inside parenthesis after class name to make the class extend another class. Example (Duck extends Animal):

class Animal:

…

class Duck(Animal):

…

* + No special tags needed (e.g. @Override) to override a method.
  + Accessing the superclass: super()
* Applying polymorphism to classes
  + Can traverse through an iterator of objects of different classes, and call a method that each class implements differently
  + Can pass any type of parameter into a function as long as it implements the methods and data needed by the function.
  + This can be done because Python is loosely typed