Python Programming Introduction

Instructor – Glenn Downing

Website: oopl.com/jpl

Repository on github.com: gpdowning/oopljpl

*Day 1:*

Many different ways to run a python program – see Hello.py on repo.

Python does have types internally, but you do not have to declare types when using variables.

Indentation is important! Colons indicate the start of a new indentation block. Spaces or tabs are ok, but just be consistent.

Also, note that there are no semicolons at the ends of lines; one statement per line.

Division: “//” indicates integer division, ‘/’ is floating point division.

If running a script within an interpreter, need to include a guard around main():

if \_\_name\_\_ == “\_\_main\_\_” :

main()

Assertions.py:

It is advantageous to use assert statements at the start and the end (at least) of functions to specify what must be true when entering and exiting a function.

Optimization with –O flag turns assertions off. It is useful to include asserts, but it is not useful to use asserts to test code for precisely this reason.

Another advantage is that they are self-documenting safety nets that informs the reader of intent.

Limitations:

1. Use for preconditions and postconditions, not testing code.
2. They are good for finding programmer error mistakes, not user input or interaction. Its fine for argument checking, as long as the source of input is not the user.

So what should we use to test code? Python has a library called unittest. This library has testing functions that will run all code, even if assertions fire.

To ease usage of certain functions, you can use the syntax: from [library] import [item]. This allows you call [items] directly in the code, rather than as [library].[items]

Coverage1.py:

Glenn advocates obtaining a tool called coverage. See code for URL. It will report on what percentage of lines are run during execution.

Exercise 1 – factorial unit tests and function:

Recursive procedure – A function that calls itself.

Recursive process – a recursive action that can be turned into something iterative (aka a loop).

Some compilers (e.g. lisp) will automatically optimize recursive processes into loops. This saves on stack space execution. However, python does not do this.

“reduce” is a clever way to apply a binary operation to a range of numbers.

A syntax for python internal object members is “\_\_[member]\_\_” For example, print([function].\_\_name\_\_) will print out the name of [function].

“timeit” is a useful function to measure execution time of a function. Need to feed strings describing commands (e.g. functions, arguments, import statements) it needs to run the function.

“range” is an example of a python generator – intuition might tell us that it’s building a sequence, but really it’s building backend machinery that produces numbers one by one on demand.

Handling user error:

Ways to communicate between functions:

1. Return values
2. Use of an additional input variable that the called function alters (be careful of pass by value v. pass by reference)
3. Global variables
4. Throw exception

When things complain (e.g. a function), we have a range of options:

1. Ignore them
2. Halt execution (e.g. using assert())
3. Raise an alert that demands action – this is called an exception

Mechanism for (3) involves try-catch (possibly multiple catches per try) pairs in conjunction with the “raise” keyword. Several possibilities exist when you hit a try-catch:

1. No exception to catch – keep going
2. Exception triggered, but it isn’t the same one as the one specified by the catch – stop execution and go to the calling function (to look for a a proper catch)
3. Exception triggered and it is specified in the catch – stop execution and handle exception
4. No try-catch in caller – pass to caller’s caller for possible handling.

In python, the equivalent to the “catch” keyword is “except”

“tuple” is an immutable list type – once created, type and size will not change. To create a one element tuple, add a trailing comma: e = (3,) creates an 1-element tuple with the value 3.

Distinction between “is” and “==” operator – “is” is a reference comparator (compares whether two objects share the same space in memory), whereas “==” compares values.

Exploring Types:

Tuples are immutable – size of the tuple in memory doesn’t change. In fact, what the tuple holds are memory address (pointers) to its elements. More

Lists are

Tuples and lists do not care about contents, but sets do. Sets have 2 limitations:

1. A set cannot have duplicate elements.
2. Elements in the set must be “hashable” – they must be immutable! A list cannot be an element of a set, but we could put a tuple or string into a set.
3. Set implementation is as a hash table, leading to constant time lookup. Adding things to a set can occasionally be expensive (due to rehashing).

A dictionary is a set with each element consisting of two parts: [key] : [value]. A key must adhere to the rules of a set’s elements; corresponding values could be anything. Implementation is again as a hash table, leading to constant time lookup on key values, but linear time lookup on values.

Constructor function is denoted as \_\_init\_\_(self, [args]). Be sure to call constructor of any base types you are inherited from.

Oddity: The type FunctionType (aka the type for functions) is not a builtin type. To use, must use “from types import FunctionType”

Operators:

lvalue and rvalue designations are given by whether an item can be placed on the left or right side of an assignment.

All python operators always produce rvalues.

Variety of builtin operators – bitwise (&,|,^), Boolean logical (and/or/not), overloaded arithmetic (result can depend on operands – using + on strings leads to concatenation, on lists or tuples also leads to concatenation)

Assignment needs to be considered with care. The “thing” stored by a variable can be:

1. A value: x = 3, x holds (“is”) the value 3
2. A memory address: x = [2, 3, 4], x holds an address to the list [2, 3, 4].

What are the consequences of this? Consider the following:

A = [2, 3, 4]

B = A

print(A is B) # this will give true!

A += [5]

print(B) # this will give [2, 3, 4, 5] since A and B point to the same memory address

However, what if we try this with an immutable data type like a tuple?

A = (2, 3, 4)

B = A

print(A is B) # this will give true!

A += (5,) # hmmm. Since the original tuple (2, 3, 4) cannot be modified, what occurs is that python creates a new tuple (2, 3, 4, 5) and assigns the address of the new tuple to A

print(A) # gives (2, 3, 4, 5)

print(B) # gives (2, 3, 4)

print(A is B) # gives false! A no longer points to the same tuple

Consequently, conclude that the nature of assignment depends on the rvalue being assigned.

Iterability differs from indexibility; interability is the capability of elements in a data structure to be traversed in no particular order. Indexibility is interability with a specific ordering, and is stronger than interability.

Iterators – an object used to iterate over an object.

*Day 2:*

Variables.py:

Colon operator on lists; like matlab, except right-hand index is again noninclusive. It will also allow you to make a copy of a mutable object. For example:

a = [2, 3, 4]

b = [:]

assert a is not b # this will pass!

assert a == b # this will also pass

Sidebar – python will not let you make a copy of an immutable object.

+= operator, applied to lists, only require an iterable item as an rvalue; however, + applied to lists requires another list. More generally, a += N is equivalent to a = a + N for immutable objects, but they are not equivalent for mutable objects (a + N will create a new object, whereas a += N will not)

Cache.py:

Once you go past 256 (1 byte) or below -5, then variable no longer stores the value, it stores the address. If you change a variable to within this range, it will internally restore it to being stored locally. The reason for this is performance – apparently, statistically this range gives optimal performance.

If the interpreter sees the same exact value assignment twice, it will point the variables to the same place.

When new objects are created is largely a function of mutability of the types involved.

Arguments.py:

Mutability/immutability affects how arguments are passed into a function. As a method for communicating back to a calling function, when using arguments, one must ensure that the argument is both a type that will be passed by value (since no facility exists in Python for specifying an object reference) and is immutable (tuples are passed by reference, but they are immutable, so it is an effective pass by value)

Iteration.py:

Certain functions can be called on objects in python because objects have certain attributes. For example, you can call iter() on a list because the list has the attribute \_\_iter\_\_.

Iterating over a dictionary by default iterates over the keys.

Iterating over sets and dictionary keys do not guarantee any specific order.

For loops can have an else clause, but the else clause runs when the for loop terminates **normally**. It may be useful if the for loop is looking for something, finds it, and breaks with the break keyword (else clause doesn’t run), or when the loop doesn’t find something and terminates normally (else clause runs).

The count class is a type of iterator.

Invoking iter() on an iterator returns the iterator itself. This is useful when dealing with multiple loops going over the same range, but want to maintain continuity between loops:

x = count(0) # 0, 1, 2, ...

assert type(x) is count

assert hasattr(x, "\_\_next\_\_")

assert hasattr(x, "\_\_iter\_\_")

assert not hasattr(x, "\_\_getitem\_\_")

#assert (x[0] == 0) # TypeError: 'itertools.count' object is not indexable

s = 0

for v in x :

if v == 10 :

break

s += v

assert s == 45

for v in x :

if v == 20 :

break

s += v

assert s == 180

To be specific, to be iterable partially means that calling iter() on an object results in a new iterator.

Once an iterator has iterated over all elements, it will become exhausted and not return new elements.

You can put an expression inside of brackets for list construction. This is called list comprehension:

x = [2, 3, 4]

y = [v \* 5 for v in x] # list comprehension

assert type(y) is list

assert not hasattr(y, "\_\_next\_\_")

assert hasattr(y, "\_\_iter\_\_")

assert x == [2, 3, 4]

assert y == [10, 15, 20]

Functions are actually defined via lambda calculus. For example, the function add(x, y) is more formally written:

add = lambda x, y: x + y

This is the assignment of a function to the name “add.” We can even call the function directly without a name:

x = [2, 3, 4]

y = map(lambda v : v \* 5, x)

assert type(y) is map

assert hasattr(y, "\_\_next\_\_")

assert hasattr(y, "\_\_iter\_\_")

assert x == [2, 3, 4]

assert list(y) == [10, 15, 20]

assert list(y) == []

map() produces an iterator. Interestingly, iterators are lazy; they are doing the minimum necessary to create the next item requested when next() is called. In the above, the lambda function is called when an item is requested. This behavior is good when you don’t require all the elements of x most of the time, bad when you usually need all the elements of x. It also suggests that if x is mutable, then what y produces is dependent on what x is at the time next(y) is called.

Generator v. map v. iterator? Generators produce iterators.

List comprehension and its variations (generators) can be used in numerous ways to create new elements succinctly. For example:

x = [2, 3, 4, 5, 6]

y = []

for v in x :

if v % 2 :

y += [v \* 5]

assert x == [2, 3, 4, 5, 6]

assert y == [ 15, 25]

can be condensed to:

x = [2, 3, 4, 5, 6]

y = [v \* 5 for v in x if v % 2]

assert x == [ 2, 3, 4, 5, 6]

assert y == [ 15, 25]

List comprehension syntax can also be applied to sets and, in this case, is called set comprehension. Adding elements to a set is done using the union operator |= (so that only unique values are retained).

Dictionary comprehension also is possible!

The functions filter, zip, and map can aid in iteration.

Functions all() and any() are designed to return Booleans if everything in the iterable argument is true (all) or at least one element is true (any).

*Day 3:*

It may be possible to optimize code by using a lambda function to reduce (or eliminate entirely) the amount of storage needed to keep intermediate results.

Best use case for a generator is for cases where you don’t intend, on average, to exhaust the generator. Otherwise, it is debatable whether it is faster to make a list or use a generator.

Assignment.py:

Multiple variables can be assigned on one line. This is called parallel assignment:

i, j = 2, 3

assert i == 2

assert j == 3

We can also enclose the left-hand side with either () or [], but this does not imply that the LHS is either a tuple or list. This is merely an alternative syntax for parallel assignment.

No temporary swap:

i = 2

j = 3

i, j = j, i

We can have a function demand an iterable of a certain length by using ():

def f((x, y), z) :

This function insists that the first argument must be an iterable of length 2.

FunctionKeywords.py:

Named assignment is possible in lieu of positional assignment. The two can even be mixed, as long as all the positional assignments are done before named assignments.

def f (x, y, z) :

return [x, y, z]

assert f(2, 3, 4) == [2, 3, 4]

#f(2, 3) # TypeError: f() takes exactly 3 arguments (2 given)

#f(2, 3, 4, 5) # TypeError: f() takes exactly 3 arguments (4 given)

assert f(2, z = 4, y = 3) == [2, 3, 4]

#f(z = 4, 3, x = 2) # SyntaxError: non-keyword arg after keyword arg

#f(2, z = 4, x = 2) # TypeError: f() got multiple values for keyword argument 'x'

#f(2, z = 4, a = 5) # TypeError: f() got an unexpected keyword argument 'a'

Caution – positional assignments must use the names provided by the function. If the name of those arguments change, then the function call will fail if they are not altered accordingly.

FunctionDefaults.py:

Similar to how positional assignments must precede named assignments, required arguments in a function prototype must precede default assignments:

#def g (x, y = 3, z) : # SyntaxError: non-default argument follows default argument

# return [x, y, z]

def g (x = 2, y = 3, z = 4) : # everything has a default

return [x, y, z]

Setting a default to a mutable object is dangerous because the defaulting only occurs once! Consequently, mutable defaults have memory that can happen from call to call when the default is used. Usage of the keyword None (equivalent to NULL in C) with appropriate checking can help alleviate this issue.

FunctionUnpacking.py:

Unpacking only works in function calls.

The \* operator can be used to unpack an argument by placing it immediately before the argument in the function call. This places the elements (rather than a tuple/list/iterable) into the arguments. Unpacking and named assignment have no particular priority, but you can only unpack one thing once, and both must occur after positional assignment.

Order of evaluation is by position, then unpacking, then by name. Unpacking will default to the earliest unassigned arguments.

The \*\* operator can be used on dictionaries (only) to unpack values where the keys are names of the function’s arguments. This is equivalent a named assignment. Dictionary unpacking must occur after any positional or named assignments.

\* unpacking and positional assignment must come before named assignment, which must come before \*\* unpacking

FunctionTuple.py:

Can also use \* in function definitions as applied to the parameters. This allows functions to take an undefined number of arguments. When applied on function definition parameters, they pack arguments into a tuple (and only a tuple)

Packed parameters (preceded by \*) cannot be found via a dictionary unpacking.

FunctionDict.py:

Can also use \*\* in function definitions as applied to the parameters. If assigning manually to a \*\* argument, need to use key = value syntax.

Creating our own types:

Unlike other languages, we can keep adding data to the class instances even after the instance has been created. This can be done using the . notation. Start with an empty class A:

x = A()

y = A() # y contains no data

x.i = 2

x.j = 3 # x now contains 2 pieces of data. y still has no data

Constructors cannot be overloaded. Constructors dictate what users should give you to build a class instance.

In python, methods are simply treated as functions where the first argument is the object itself.

Default arithmetic operators defined and overloadable via \_\_add\_\_, \_\_sub\_\_, etc. In-place equivalents (e.g. +=) map to functions such as \_\_iadd\_\_

conditional expressions – a quick way to decide what to insert. For example, see **bold** portion below:

def \_\_str\_\_ (self):

return ("(" + str(self.real) + **("+" if self.imag >= 0 else "")** + str(self.imag)+"j)")

What is inserted is a string, not a tuple, or a list.

Sidenote - \_\_str\_\_ implicitly called by print(). This means that if \_\_str\_\_ is defined for a class, you can call print() on the instance of the class.

*Day 4:*

FileInputOutput.py:

Need to import sys module to get OS interactions. Commandline arguments get placed into argv[]. In python, argv does include the name of the script.

Text files, when imported using open(), result in an iterable handle where each iterable element is a line. This is what the file iterator’s next() behavior is.

print()’s natural behavior is to add a newline after each line printed out. This can be suppressed by calling print() with a second argument as print(str, end = “”). This places an empty string at the end of every line printed. Default behavior of print is equivalent to print(str, end = “\n”)

Classes can be specified inside of another class, but this primarily affects naming, not access. If a class B is specified inside of class A, you can create an object by x = A.B(). No object A is constructed; only an object of type B is made.

Keyword yield: similar to return in that it leaves the function and returns a value to the caller. However, when the function is called again, the function keeps going from the last yield that was used. Additionally, a function with yield does **not** run when called! Instead, what is created and returned is an object that can be iterated over. When next() is called on the returned object, then the function runs, up to the next occurrence of yield.

Range2.py

Rewrite range without the embedded iterator class. Use yield.

Map v. generator – generators have the ability to accept multiple for statements and if statements (filters). On the other hand, map() can take a n-ary function with the appropriate number of iterables and feed each of them in lock step to the function

GlobalVariables.py:

Global variables are specified as top-level statements outside of function definitions as assignments.

If we only ever want to read a global, we just need to declare it. If we want to modify it locally, we need to declare it as global using the global keyword inside the block (e.g. “global v5”).

Class global variables are created by specifying variables via assignment within a class, but outside any class methods. These variables are shared among all instances of the class. To access outside the class, need to preface the variable name with the class name (e.g. “A.v0”). However, within the class, there is no need to precede the variable name with the class name; if done, this will in fact give an error.

Internally, class global variables are stored as a dictionary. The dictionary has a name \_\_dict\_\_, with the keys being the variable name and the values being the corresponding variable values.

You can delete class global variables with the del keyword.

Python does not have the C++ notion of public/private data members (i.e. encapsulation). Instead, python provides a weak way to show programmer intent to “hide” variables. This is done via preceding any variable desired to be private by \_\_. Note, however, the member can still be accessed as \_[class]\_\_[var]. Finally, inside the class, you may refer to the “private” variable by either convention.

InstanceVariables.py:

We can specify instance variables by using self.[var] assignments inside the class definition but outside any methods. Note that each instance has its own \_\_dict\_\_ that stores the instance’s instance variable as key-value pairs.

If there is an instance and class variable with the same name, how you get to each one depends on how you access them.

When you add a variable to an object instance outside a class definition, that new variable is unique to that object instance:

assert not hasattr(x, "v5")

#assert x.v5 == 4 # AttributeError: 'A' object has no attribute 'v5'

#assert x.\_\_dict\_\_["v5"] == 4 # KeyError: 'v5'

x.v5 = [2, 3, 4]

assert hasattr(x, "v5")

assert not hasattr(y, "v5")

assert x.v5 == [2, 3, 4]

assert x.\_\_dict\_\_["v5"] == [2, 3, 4]

y.v5 = [2, 3, 4]

assert hasattr(y, "v5")

assert y.v5 == [2, 3, 4]

assert y.\_\_dict\_\_["v5"] == [2, 3, 4]

assert x.v5 is not y.v5

assert x.v5 == y.v5

Strictly speaking, there are 2 types of class-related variables: class variables and instance variables. Both types are stored in dictionaries, one at the class level (shared by all instances) and one for each instance. Furthermore, for each instance, you can add variables on the fly via the [instance].[var] = value. This will add the variable to the dictionary for that specific instance. If the newly added variable has the same name as a class variable, it will shadow the class variable if access is made using the [instance].[var] syntax.

Methods.py:

You can create instance variables inside methods, but not class variables.

Instance methods are distinguished by the presence of the keyword self as the first argument. Class methods do not have self as the first argument and their definition must have the decorator @staticmethod in the line above the definition.

Closure.py:

Closures: the concept that a function keeps a copy of its local variables, even if the calling scope appears to indicate that the local variable should not be accessible. Succinctly, closure refers to the fact that python functions can have state.

Sequences.py:

Negative indices index from right to left. Rightmost element is at index -1 (not 0!). Using negative indices doesn’t change anything in the direction of iteration; each element has both a positive valued and negative valued index.

Interesting case: a[: : -1] reverses all the element of list a.

str() converts a non-iterable input into a string. If the input is iterable, then it attempts to concatenate the elements into a string, which implicitly mandates that all the elements be strings themselves.

Lists.py:

append() adds elements to the end of a list one at a time. extend() adds a group (i.e. another list, tuple, set) to the end of the list.

pop() removes the last item from a list. remove(x) removes an (or possibly all) elements of value x from a list.

Be careful with replication operator \*: replicating lists does not create new lists when replicating; it merely replicates pointers.

One can use slices as a lval. See lists.py for examples, and note how application with step size differs.

Strings.py:

Strings may be delimited with ‘ or “ or ‘’’ or “””. Prepending a single-quoted string with ‘r’ causes it to count all characters, including delimiting \.

in, upper(), lower(), find(), replace(), and split() are all useful string functions (among others used in the strings.py file).

RegExps.py:

‘.’ operator does not match the newline \n character.

Learning regular expression facility of an editor is invaluable.

*Day 5:*

FormattedOutput.py:

Format specifiers analogous to those used in Fortran or C. Output formatting binary operator is %, with an output formatting string on the LHS and an object or tuple.

Sets.py:

set() creates an empty set.

Comparison operators used on sets indicate proper subset (< >) or improper subsets (<=, >=).

Sets cannot have duplicate values, but be careful since python imparts/interprets values interestingly. For example, True == 1 and 2.0 == 2, so if a set is constructed with {True, 1}, it is ambiguous as to whether you will get {True} or {1}, but you will **not** get {True, 1}

Dicts.py:

Dictionaries can be built over iterables that are key-value pairs – iterables over iterables of length 2 where the first element is hashable.

If you create a dictionary with duplicate keys, python will take the last key-value pair of the duplicate keys.

{} creates an empty dictionary.

You can pop([key]) on dictionaries. pop() will return the associated value and remove the key-value pair from the dictionary.

Unlike direct indexing using [], using the get() method on a dictionary does not return an exception if the invoked key does not exist. Instead, get() will return None.

zip() is a useful function to create key-value pairs (via a lazy iterator) that may be converted to a dictionary. If arguments to zip are unequal, it will operate according to the shortest input.

For a class, you need \_\_str\_\_ function defined to format the output; otherwise, what will be printed is the address of the instance.

Decorators.py:

A means to integrate checks into a function via the use of closure on functions. More than one decorator may be applied to a function, and decorators can even take arguments themselves.

More properly, a decorator accepts a function as input and returns a function as output.

Inheiritance.py:

Unlike java or C, you do not have to call the parent constructor in an inherited class’ constructor; it is, however, recommended that you do so. If no constructor is defined in the inherited class, the parent constructor is called by default.