Python class - Notes

(taught by Glen Downing: http://www.oopl.com/jpl/)

Day 1: Monday, Jan 12 2015

examples/Assertions.py

- Python enforces indentation. Need to use tabs or spaces (just stick to one)

- '//' integer division, '/' is floating division

- when an assertion fails, an error is thrown

- assertion when you enter the function and when you leave the function 🡪 tells you where it failed (good for testing)

- can run program without assertions: python –O

- **useful to have asserts in code, NOT to test code (assertion crashes the program)**

- pros: program-error detection (mistake you make as programmer)

- cons: not for user-error detection (mistake user makes when running your program)

examples/UnitTests.py

- Python library unittest

- from unittest import main, TestCase 🡨 instead of import unittest where you'd have to write unittest.main and unittest.TestCase

- in the import, main is name of function and TestCase name of a class

- class MyUnitTests (TestCase) 🡨 MyUnitTests object is inheriting from TestCase object (was imported from unittest library)

- self.assertEqual 🡨 assertEqual is function inherited from TestCase

examples/Coverage1.py

- dots (success) & F's (failure)

- coverage3 run --branch Coverage1.py 🡨 run coverage3 program on Coverage1.py

- coverage tests how many lines of code are run

exercises/Factorial2.py

- for loops: for x in range(0,3) 🡨 from 0 to 3-1, so for x = 0, 1, 2

- can have functions within functions

- can pass function as a parameter to a function, e.g. test(factorial\_recursion)

- t = timeit(f.\_\_name\_\_ + "(100)", "from \_\_main\_\_ import " + f.\_\_name\_\_, number = 1000)

- f.\_\_name\_\_ returns function name 🡪 e.g. calls factorial\_recursion(100)

- from \_\_main\_\_ import function name 🡨 need to know where to find the function

- number = 1000 🡨 run factorial\_recursion(100) 1000 times and time it

- range is really fast

- Python passes parameters **by value**

examples/Exceptions.py

- raise domain\_error("something wrong") 🡨 throws an exception, then can catch in try-catch

- exceptions bubble up to the top error until it's caught (good so you don't have to manually pass it up yourself), keep catching the caller

- **use try…except for checking user-error detection**, can also use else and finally

- tuple is a list type in Python, e.g. t = (2,3)

- if you want one-element tuple, need to put in trailing comma, e.g. s = (2,) 🡨 (2) would declare a variable with value 2

- is for memory address comparison, and == is a content comparison

examples/Types.py

- tuples store addresses, are immutable (once declared, cannot modify)

- lists are mutable (can change)

- sets care about content:

- can't have 2 of the same value in a set

- cost of lookup: constant (implemented with hash table, run function on value 🡪 address 🡪 if multiple values at address, linked list at that address)

- limit: content has to be hashable, not mutable (can't put list/set/dict in a set)

- dictionary has pairs of items (key, value); hash table, values must be hashable

- looking up key is very fast, looking up value is not

- class A : 🡨 building own type; def \_\_init\_\_ (self, i, f) 🡨 constructor

- class A (B) : 🡨 define type A with parent B, A calls B's constructor

- if you want to confirm a FunctionType, must from types import FunctionType

- // does floor division

- can do assert i < j < k < l

- + can concatenate lists & tuples, \* can replicate lists & tuples

- a = 2 🡨 a's value is 2 vs. a = [2] 🡨 a is the address where the list lives

- for in requires iterability, not indexability (more flexible)

Day 2: Tuesday, Jan 13 2015

examples/Variables.py

a = [2, 3, 4]

print(a[1:2]) # [3]

print(a[1:3]) # [3,4]

- using colon makes copy of mutable (list, set) object, not pointing to same mutable object (doesn't work for tuples), e.g. a[:] makes a copy of list a

- lists += x only cares that x is iterable

examples/Cache.py

- if storing an int > 256, it stores the variable as an address (e.g. x = 257), cache: [-5, 256]

- floats are always addresses

- strings are immutable, so whenever you concatenate a string, it makes a new string

examples/Iteration.py

- assert not hasattr(a, "\_\_next\_\_") 🡨 the next function cannot be called on list a

- assert hasattr(a, "\_\_iter\_\_") 🡨 the list a is iterable

- when passing parameters, if you want them to be changed by the function, the parameters must be references and mutable

- list += is very accommodating, the added value just needs to be iterable

- can have more than one variable in a for, e.g. for u, v in a 🡨 can iterate through iterable list of objects with 2 elements like a = [[2, "abc"], [3, "def"], [4, "ghi"]]

- dictionaries are like associative arrays, when looping don't know what order dict will iterate

- a range is its own type/object (not a list or tuple), and immutable

- list(x) 🡨 x must be iterable, e.g. list(range(10)) 🡨 build list using range to populate it

- for loops can have else clause which runs when it terminates normally (don't break out early)

- iterators respond to next, but also to iter (returns itself)

- count(0) is an infinite sequence starting with 0. for v in x multiple times just continues counting

x = count(0) # 0, 1, 2, …

for v in x :

print v

if v == 10 :

break

# 0, 1, 2, … 10

for v in x :

print v

# 11, 12, 13, …

a = [2, 3, 4]

p = iter(a)

for v in p :

print v # 2, 3, 4

for v in p :

print v # nothing! (already exhausted iterator)

x = [2, 3, 4]

y = **[v \* 5 for v in x]** # [10, 15, 20]

- add = lambda x, y: x + y introduces a function on the fly (def add (x, y) : return x + y)

- map is an iterator and is exhaustible

- **list comprehension**: concise way to create lists

- iterators are lazy, map has eager list comprehension so it's only built when called

- map is better to call with immutable (tuple) instead of mutable (list) so you don't accidentally change the list

- |= 🡨 in-place union for sets (operator); "union-equal", only add if not already in the set

- zip() can take indefinite number of arguments; transposed a matrix

Day 3: Wednesday, Jan 14 2015

exercises/StDev2.py

- assertRaises() 🡨 checks for raised exceptions

examples/Assignments.py

- can do parallel assignments, e.g. i, j = 2, 3 🡨 on the right, must be an iterable of 2 elements

- nice swap: i, j = 2, 3 i, j = j, i

examples/FunctionKeywords.py

def f (x, y, z) : return [x, y, z]

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

- when calling a function, can set arguments by calling them by name

- must do positional assignments first, then calling by name

examples/FunctionDefaults.py

- can set defaults in function definition, at end of arguments 🡪 def (x, y, z = 4)

- defaults only happen once, so want to use immutable defaults (it keeps updating the default)

- None is equivalent to null

examples/FunctionUnpacking.py

- t = (3, 4) \*t 🡨 **one star** **unpacks** an iterable to 3, 4; can only unpack one thing once

- order for arguments in function: by position, by unpacking (\*), by name & dictionary unpacking (\*\*)

- for dictionaries, .keys(), .values(), and .items() return a generator

- \*\*d 🡨 **double star only unpacks dictionaries** and returns values

def (x, y, z) : return [x, y, z]

d = {"z" : 4, "y": 3, "x" : 2}

assert f(\*\*d) == [2, 3, 4]

- the function sets x, y, z to the x, y, z values of the dictionary

examples/FunctionTuple.py

- functions can take an indefinite number of arguments, e.g. def f (x, y, \*z) 🡨 take at least 2 arguments and put the rest in z

examples/FunctionDict.py

- \*\*d in function definition expects key value pairs, e.g. f(a = 4, b = 5) or \*\*d

Classes

class A :

def \_\_init\_\_ (self, i = 0, j = 0) :

self.i = i

self.j = j

def f (self) :

return self.i\*\*2 + self.j\*\*2

def g (self, k) :

return self.f() + k\*\*2

# return A.f(self) + k\*\*2

x = A(2, 3)

y = A()

A.f(x)

a = [A(2, 3), A(4, 5), A(6, 7)]

m = map(lambda x : x.f(), a) # x stores each A()

m = map(lambda x : A.f(x), a) # same as previous line

m = map(A.f, a)

- all class functions must start with self as first argument

- print(len(x)) # print(x.\_\_len\_\_())

- if \_\_name\_\_ == "\_\_main\_\_" : main() 🡨 if importing this script in the python interpreter, let the functions be available to the interpreter but don't run the script

Day 4: Thursday, Jan 15 2015

examples/FileInputOutput.py

- sys.argv[0] 🡨 grabs first program argument

- files are iterable, e.g. f = open([filename]) for v in f : print (v, end="")

🡨 do not print a carriage return at the end of the print statement since the file already has \n's in it (end = "")

Class within a class, e.g. this is an interable (e.g. inside List, has a class Iterator):

class A :

class B :

def \_\_init\_\_ (self, j) :

self.j = j

def g (self) :

return self.j

def \_\_init\_\_ (self, i) :

self.i = i

def f (self) :

return self.i

def h (self) :

return A.B(self.i + 2) # build and return a B object (return a new Iterator)

x = A(2) print(type(x)) # A print(x.f()) # 2

y = x.h() print(type(y)) # A.B print(y.g()) # 4

examples/Generators.py

- yield is like return in a function, but it remembers where it was and when you call the function again, it continues after the yield

def f () :

print "hi"

yield 2

print "bob"

yield 3

print "alice"

yield 4

print "done"

x = f() # this does not call f(), it returns an iterator object

print(type(x)) # generator (a type of iterator)

i = next(x) # "hi"

print(i) # 2

y = f()

print(type(y)) # generator

j = next(y) # "hi"

k = next(x) # "bob"

print(k) # 3

k = next(x) # "alice"

print(k) # 4

k = next(x) # "done" -> StopIteration

- map can take any number of iterables and use that many variables in the provided function

examples/GlobalVariables.py

- **global variables cannot be modified in functions, unless declared so first**, e.g. global a

- if just reading global variable in a function, don't need to declare it global (just use it)

examples/ClassVariables.py

- when you create a class variables, you add those variables to a dictionary (key = var name, value = variable value)

- Python doesn't really have private class variables, just adds "\_\_" to the front of private variables. You can still grab those variables outside the class though (\_[class name]\_\_\_[var name])

- can't create quasi-private variables (\_\_) from outside the class

- outside a class, you can assign new attributes/variables, e.g. A.v5 = [2, 3, 4]

- can delete attributes, e.g. del A.v0

examples/InstanceVariables.py

- instance variables use self (created for each instance, vs. class variable created once for all instances)

examples/Methods.py

- @staticmethod creates a class method (can't access instance variables, i.e. self)

- instance methods have self parameter

examples/Closures.py

- when you build functions within functions, you're in a certain scope and has access to the local variables you mentioned

- functions are given parameters and return values, so functions do not have state. Classes have state. A closure is a function with state.

examples/Sequences.py

- can use negative indices in Python, e.g. a = "01234" assert a[-4:-1] = "123"

examples/Lists.py

- can assign a slice of a list, e.g. a[1:4] = [0, 0] (doesn't have to be same length on both sides of '=')

Day 5: Friday, Jan 16 2015

examples/FormattedOutput.py

- %7.2f 🡨 7 space output with 2 decimal places, **formatting does rounding**

examples/Sets.py

- frozensets are immutable sets

- seeing if member in a set is fast operation (vs list)

- sets use many operators that do common set functions: | & - ^

examples/Dicts.py

- can use zip() to put 2 tuples together and make a dictionary

examples/Decorators.py

def pre\_gtz (f) :

def g (n) :

assert n > 0

return f(n)

return g

@pre\_gtz # decorator for the function

def cycle\_length (n) : …

print(cycle\_length(5)) # 6

print(cycle\_length(0)) # fail

- with decorators like @pre\_gtz, you can move the pre-condition assert n > 0 taken it out of the function cycle\_length()