Advanced python stuff

# Chapter One (Introduction)



Match statement

* Cases use pattern matching
* <https://mathspp.com/blog/pydonts/pattern-matching-tutorial-for-pythonic-code>
* <https://www.python.org/dev/peps/pep-0636/>
* Switch statement on steroids

Variables

* Are pointers, each variable points to an object that holds a value
* All objects that variables point to have unique id’s

Fstring Format Descriptors

* f
  + floating point
* g, G
  + Uses e or E for exponents less than -4 or greater than precision
* s
  + String or any object with a \_\_str\_\_() method
* r
  + Same string as repr()

str.format()

* Provides positional formatting capabilities
* r = "{0} {1} {2}".format('GOOG',100,490.10)

r = "{} {} {}".format('GOOG',100,490.10) # Automatically advances

r = "{name} {shares} {price}".format(name='GOOG',shares=100,price=490.10)

r = "Hello {0}, your age is {age}".format("Elwood",age=47)

r = "Use {{ and }} to output single curly braces".format()

stock = {'name' : 'GOOG',

'shares' : 100,

'price' : 490.10 }

r = "{0[name]} {0[shares]} {0[price]}".format(stock)

x = 3 + 4j

r = "{0.real} {0.imag}".format(x)

r = "{name:8} {shares:8d} {price:8.2f}".format(name="GOOG",shares=100,price=490.10)

name = "Elwood"

r = "{0:<10}".format(name) # r = 'Elwood '

r = "{0:>10}".format(name) # r = ' Elwood'

r = "{0:^10}".format(name) # r = ' Elwood '

* r = "{0:=^10}".format(name) # r = '==Elwood=='

Id

* gets the id of the object that a variable is pointing to

Range

* returns a range object
  + >>> r = range(10, 101, 10)
  + >>> r
  + range(10, 101, 10)
  + >>> list(r)
  + [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
  + >>> list(r)
  + [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
  + >>> sum(r)
  + 550
  + >>> sum(list(r))
  + 550
* Can be iterated but is not consumed

repr()

* yields a string that represents (

Slices

* s[start : stops : step]
* stop position is not included
* negative indexes exist

Shallow copies

* variable does not point to the same object that it is copying from
* ex. a = b[:] shallow copy
  + changing b does not change a

Deep copy

* variable is pointing to the same object that it is copying from
* import copy

c2 = copy.deepcopy(c1)

* c2 now points to the same object as c1

Sequence types

* string
* list
* tuple

Zip

* take two lists and put them into one tuple pairing up like indexes

All

* && an entire list

Any

* || an entire list

# Chapter Two (Tuples and Lists)

Lists

* Expandable arrays
* Not linked lists

List Methods

* append
* extend
* insert
* remove
* pop
* revers
* clear
* copy
* index
* sort
* +
* +=

List Comprehensions

* [<expression><input-sequence><optional condition>]
  + ex. [x\*x for x in range(1,5)]
* A faster way to code in a shorter amount of lines

Tuples

* An immutable sequence
* Special syntax for tuple of length 1
  + t = (‘a’,)

Tuple operations

* count
* index
* tuple assignment
  + a,b,c = 1,2,3

Sequence Replication

* you can multiply a sequence by an integer to replicate it

Memory and Variables

* Many variables can refer to the same object
* Python keeps a reference count for all objects
* When the reference count reaches zero, the object can be garbage collected

# Chapter Three (Dictionaries and Sets)

Dictionaries

* A collection of <key, value> pairs
  + key is unique and must be immutable
  + can’t be a list
  + can be a tuple
* dict()
  + can convert sequences of pairs
  + [(1,2),(3,4)]

Dictionary Methods

* clear
* copy
* fromkeys
* get
* in
* items
* keys
* pop
* popitem
* setdefault
  + if key is not in dictionary, add key to dictionary
* update
* values

# Chapter Four (Functions)

Variable-Length Arguments Lists

* using an asterisk before the parameter name makes the arguments collects as a tuple
  + def avg(\*stuff)
  + avg(1,2) makes stuff = (1,2)

Unpacking Tuples in a Function Call

* you can use an \* to ‘unpack’ a tuple in a function call
* >>> def f(a, b, c):

... print('a =', a, 'b =' , b, 'c =' , c, sep=' ')

...

>>> stuff = ('one', 2, 3.33)

>>> f(\*stuff) # same as f('one', 2, 3.33)

a = one b = 2 c = 3.33

Variable-Length Keyword Arguments

* You can use an \*\* to collect parameters into a dictionary argument
  + def f(\*\*kwargs)

f(a = 1, b = “two”)

* + - {‘a’ : 1, ‘b’ = “two”}

Function Attributes

* Functions can have attributes
* >>> def f():

...     "A function that returns the next counting number"

...     f.x += 1

...     return f.x

...

>>> f.x = 0 # f must have been previously defined

>>> f.\_\_doc\_\_

'A function that returns the next counting number'

>>> for i in range(5):

...     print(f(),end=' ')

...

1 2 3 4 5

>>> print(vars(f)) # vars returns attributes in a dict

{'x': 5}

Decorators

* Is a function that wraps another function with new behavior
* Can be composed to have many layers of added functionality
* def trace(f):

def wrapper(\*args1, \*\*args2):

print(f.\_\_name\_\_,'with',args1,args2)

return f(\*args1, \*\*args2)

return wrapper

@trace

def foo(parm):

print(parm)

@trace

def bar(parm1, parm2):

print(parm1, parm2)

Closures

* Contains a reference to the actual function body plus the non-local variable it uses

nonlocal statement

* tells a nested function to use the outer function’s variable rather than its own

Generators

* A function that contains a yield expression
  + used for iteration
* Calling the generator the first time returns an iterator
* Calling next() on the iterator will advance the function to the next yield expression in its code
* The function resumes where it left off from the previous call to next()

Coroutines

* Like generators, but can also receive input
  + Mainly used for consuming input only, but sometimes used to produce an on-going accumulated result
* The caller of the coroutine sends input to the coroutine

Lambda Expressions

* Use lambda keyword
* Can only take a single expression
* A return is implied

>>> words = ['hello', 'jello', 'Hello', 'Jello']

>>> sorted(words)

['Hello', 'Jello', 'hello', 'jello']

>>> sorted(words, key=str.lower)

['hello', 'Hello', 'jello', 'Jello']

>>> sorted(words, key=lambda x: (str.lower(x), x))

['Hello', 'hello', 'Jello', 'jello']

map

* builds a new iterable applying a function to each element

filter

* Extracts elements that satisfy a predicate (a boolean function)

deleter

* rather than returning a value you can delete a value
  + del ‘variable’

# Chapter Five (Classes)

Slots Attribute

* \_\_slots\_\_ = [‘*member*’, ‘*member*’]
* creates a list of all the fields that are allowed for the class
  + if anyone tries to create a field that is not in the slots list then the program throws an error

Mangled Names

* Two underscores at the beginning of the name
  + \_\_x
* Strongly discourages it from being accessed from outside the class
* To access
  + in class <class>, \_<class>\_\_x

Properties

* class Person

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

self.\_\_name = name

@property

def name(self):

return \_\_name

#\_\_given\_names

#\_\_family\_names

@property

def name(self):

return \_\_given\_names + “ “ + \_\_family\_names

Method Resolution Order

* If you have multiple inheritance class(1) will prefer methods and attributes from whichever class(2) is first in the parameters of the class(1)

Super Constructor

* super().\_\_init\_\_()
* it lets you inherit attributes from parent classes
* not needed for inheriting methods
* def \_\_init(self,name):

super().\_\_init\_\_(name)

Static Methods

* you can create static methods by using the @staticMethod decorator on your methods
* no self
* static methods cannot access any class features outside of its own scope

Operator Overloading

* \_\_call\_\_,\_\_add\_ etc.
* makes an operator do a specific action when used on the class

Abstract Classes

* To make an abstract class, make the class inherit abc from ABC
* @abstractmethod
  + used to make methods in an abstract class abstract
* Classes that inherit from Abstract Classes need to call its parent classes init function

in its own init function

* + def \_\_init\_\_(self):

test.\_\_init\_\_(self)

* super().\_\_init\_\_(name)
  + passes a parameter to the superclass constructor
* super().
  + used to access methods of a superclass
    - only use arguments if you are trying to explicitly call a certain superclass
  + organized by MRO algorithm

Data Classes

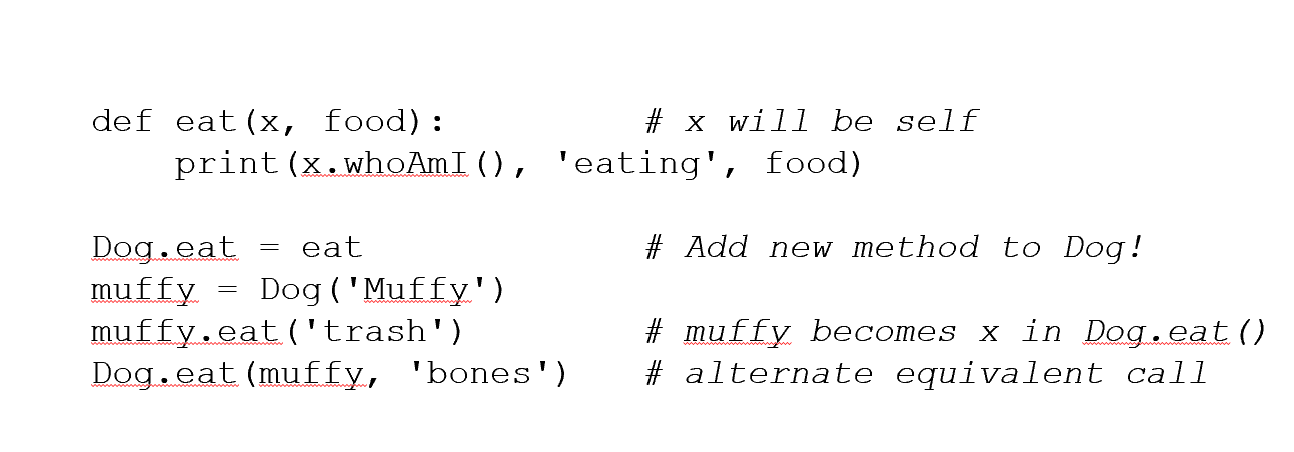
* like a C struct

vars()

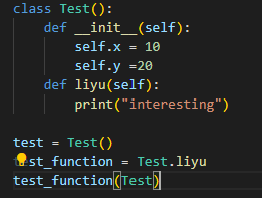
* get a dictionary of all the fields and their values in a class

Dynamically adding methods to a class

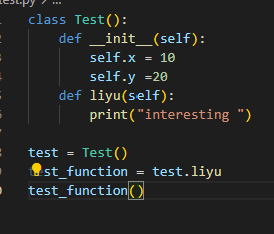
* Text

  Description automatically generated
* 

Using methods as normal functions

* You can set a variable equal to a class method and use it outside of a class instance
  + You need to pass the class as an argument tho
* 

Binding instances of a method to a variable

* 
* You can bind a variable to equal the exact function of a specific class instance

Static Methods

* @staticmethod

# Chapter Six (Modules, Packages, and Programs)

Module

* AKA a file

ChainMap

* Chains dictionaries together so they appear as one dictionary
* Text

  Description automatically generated

Counter

* counts the number of occurrences of items
* Returns a dictionary

Defaultdict

* You supply a function to construct default values

# Tkinter

# Advanced tips and tricks

Redirecting output with print

* In terminal *python main.py > “file.txt”*

For loops have else statements

* just google it

Using Tuples as Loop Variables

* stuff = [(1,2),(3,4)]

for x,y in stuff:

print(x+y)

3

7

Using \*variables to create lists

* \*variables collect all remaining data into a list
* a,\*b = range(3)
  + (0,[1,2])
* \*a,b = range(3)
  + ([1,2],3)

A Very Useful Function Pattern

* def f(\*args, \*\*kwargs):

for arg in args:

print(arg)

for key in kwargs:

print(key, '=', kwargs[key])

f(1,2,t=3,f=4)

# Stuff to do for next time

* Research repr for project C
* Research := for project C
* Use list comprehensions for project C
* Practice using copies, review slides ch.1 for project C
* Practice named tuples for project C
* use decorators for project C
* use \*,\*\* in functions for project C