pynotes

August 12, 2018

0.1 *args and **kwargs

- *args and **kwargs allow you to pass a variable number of arguments to a function.
- *args is used to send a non-keyworded variable length argument list to the function.
- **kwargs allows you to pass keyworded variable length of arguments to a function.

```
In [1]: def test_var_args(f_arg, *argv):
            print("first normal arg:", f_arg)
            for arg in argv:
                print("another arg through *argv:", arg)
        test_var_args('yasoob', 'python', 'eggs', 'test')
```

```
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

0.2 Debugging

```
In [4]: ## $ python -m pdb my_script.py
```

```
In [ ]: import pdb

        def make_bread():
            pdb.set_trace()
            return "no time"

        #print(make_bread())
```

```
In [ ]: Commands:

        c: continue execution
        w: shows the context of the current line it is executing.
        a: print the argument list of the current function
        s: Execute the current line and stop at the first possible occasion.
        n: Continue execution until the next line in the current function is reached or it retu
```

0.3 Iterators

0.3.1 Iterable

An iterable is any object in Python which has an iter or a getitem method defined which returns an iterator or can take indexes. In short an iterable is any object which can provide us with an iterator.

0.3.2 Iterator

An iterator is any object in Python which has a next (Python2) or next method defined.

0.3.3 Iteration

In simple words it is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself.

0.3.4 Generators

- Generators are iterators, but you can only iterate over them once.
- They do not store all the values in memory, they generate the values on the fly.
- You use them by iterating over them, either with a 'for' loop or by passing them to any function or construct that iterates.
- Most of the time generators are implemented as functions. However, they do not return a value, they yield it.
- Generators are best for calculating large sets of results where you don't want to allocate the memory for all results at the same time.
- next() allows us to access the next element of a sequence.
- iter, it returns an iterator object from an iterable.

```
In [ ]: def generator_function():
            for i in range(10):
                yield i

        for item in generator_function():
            print(item)
```

```
In [ ]: def fibon(n):
            a = b = 1
            result = []
            for i in range(n):
                result.append(a)
                a, b = b, a + b
            return result
        # generator version
        def fibon(n):
            a = b = 1
            for i in range(n):
                yield a
                a, b = b, a + b
```

```
In [ ]: def generator_function():
            for i in range(3):
                yield i

        gen = generator_function()
        print(next(gen))
        # Output: 0
        print(next(gen))
        # Output: 1
        print(next(gen))
        # Output: 2
        print(next(gen))
```

```
In [ ]: int_var = 1779
        iter(int_var)
        # Output: Traceback (most recent call last):
        #   File "<stdin>", line 1, in <module>
        # TypeError: 'int' object is not iterable
        # This is because int is not iterable

        my_string = "Mani"
        my_iter = iter(my_string)
        print(next(my_iter))
        # Output: 'Y'
```

0.3.5 Map, Filter and Reduce

```
In [ ]: Map applies a function to all the items in an input_list. Here is the blueprint
```

```
In [ ]: items = [1, 2, 3, 4, 5]
        squared = []
        for i in items:
            squared.append(i**2)
```

```
In [4]: items = [1, 2, 3, 4, 5]
        squared = list(map(lambda x: x**2, items))
```

```
In [ ]: def multiply(x):
            return (x*x)
        def add(x):
            return (x+x)

        funcs = [multiply, add]
        for i in range(5):
            value = list(map(lambda x: x(i), funcs))
            print(value)
```

```
In [1]: number_list = range(-5, 5)
        less_than_zero = list(filter(lambda x: x < 0, number_list))
        print(less_than_zero)
```

```
[-5, -4, -3, -2, -1]
```

```
In [6]: # Reduce is a really useful function for performing some computation on a list and ret
```

```
In [7]: product = 1
        list = [1, 2, 3, 4]
        for num in list:
            product = product * num
```

```
In [8]: from functools import reduce
        product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
```

0.4 set Data Structure

set is a really useful data structure. sets behave mostly like lists with the distinction that they can not contain duplicate values.

```
In [9]: some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

        duplicates = []
        for value in some_list:
            if some_list.count(value) > 1:
                if value not in duplicates:
                    duplicates.append(value)

        print(duplicates)
```

```
['b', 'n']
```

```
In [10]: some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
         duplicates = set([x for x in some_list if some_list.count(x) > 1])
         print(duplicates)
```

```
{'b', 'n'}
```

```
In [11]: valid = set(['yellow', 'red', 'blue', 'green', 'black'])
         input_set = set(['red', 'brown'])
         print(input_set.intersection(valid))
```

```
{'red'}
```

```
In [12]: valid = set(['yellow', 'red', 'blue', 'green', 'black'])
         input_set = set(['red', 'brown'])
         print(input_set.difference(valid))
```

```
{'brown'}
```

0.5 Set notation

```
In [13]: a_set = {'red', 'blue', 'green'}
         print(type(a_set))
```

```
<class 'set'>
```

0.6 Ternary Operators

```
In [6]: is_fat = True
        state = "fat" if is_fat else "not fat"

        fat = True
        fitness = ("skinny", "fat")[fat] #tuples
        # This works simply because True == 1 and False == 0,
        # and so can be done with lists in addition to tuples
        print("SK is ", fitness)
```

```
SK is  fat
```

```
In [7]: condition = True
        print(2 if condition else 1/0)
```

```
2
```

0.7 Decorators

```
In [9]: # Decorators are a significant part of Python. In simple words:
        # they are functions which modify the functionality of another function.
```

functions within functions

```
In [10]: def hi(name="mani"):
             print("now you are inside the hi() function")

             def greet():
                 return "now you are in the greet() function"

             def welcome():
                 return "now you are in the welcome() function"

             print(greet())
             print(welcome())
             print("now you are back in the hi() function")

         hi()
         greet()
```

```
now you are inside the hi() function
now you are in the greet() function
now you are in the welcome() function
now you are back in the hi() function
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-10-2d78395aa11b> in <module>()
         13
         14 hi()
    ---> 15 greet()


        NameError: name 'greet' is not defined
```

Returning functions from within functions

```
In [11]: def hi(name="mani"):
             def greet():
                 return "now you are in the greet() function"

             def welcome():
                 return "now you are in the welcome() function"

             if name == "mani":
                 return greet
             else:
                 return welcome

          a = hi()
          print(a)
          print(a())
```

```
<function hi.<locals>.greet at 0x7f7c64228730>
now you are in the greet() function
```

```
In [13]: def a_new_decorator(a_func):
             def wrapTheFunction():
                 print("before executing a_func()")

                 a_func()

                 print("after executing a_func()")
```

```
            return wrapTheFunction
        @a_new_decorator
        def a_function_requiring_decoration():
            """Decorate me!"""
            print("I am the function which needs some decoration")
        a_function_requiring_decoration()
        a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

```
before executing a_func()
I am the function which needs some decoration
after executing a_func()
```

```
In [14]: class logit(object):
             def __init__(self, logfile='out.log'):
                 self.logfile = logfile

             def __call__(self, func):
                 log_string = func.__name__ + " was called"
                 print(log_string)
                 # Open the logfile and append
                 with open(self.logfile, 'a') as opened_file:
                     # Now we log to the specified logfile
                     opened_file.write(log_string + '\n')
                 # Now, send a notification
                 self.notify()

             def notify(self):
                 # logit only logs, no more
                 pass
         @logit()
         def myfunc1():
             pass
         class email_logit(logit):
             '''
             A logit implementation for sending emails to admins
             when the function is called.
             '''
             def __init__(self, email='admin@myproject.com', *args, **kwargs):
                 self.email = email
                 super(email_logit, self).__init__(*args, **kwargs)

             def notify(self):
                 # Send an email to self.email
                 # Will not be implemented here
                 pass
```

```
myfunc1 was called
```

0.8 Multiple return values

```
In [15]: def profile():
             name = "Danny"
             age = 30
             return (name, age)

         profile_data = profile()
         print(profile_data[0])
         print(profile_data[1])
```

```
Danny
30
```

```
In [16]: # Or by more common convention:
         def profile():
             name = "Danny"
             age = 30
             return name, age

         profile_name, profile_age = profile()
         print(profile_name)
         print(profile_age)
```

```
Danny
30
```

```
In [17]: from collections import namedtuple
         def profile():
             Person = namedtuple('Person', 'name age')
             return Person(name="Danny", age=31)

         # Use as namedtuple
         p = profile()
         print(p, type(p))
         print(p.name)
         print(p.age)

         # Use as plain tuple
         p = profile()
         print(p[0])
         print(p[1])

         name, age = profile()
         print(name)
         print(age)
```

```
Person(name='Danny', age=31) <class '__main__.Person'>
Danny
31
Danny
31
Danny
31
```

0.9 Mutation

- In Python the default arguments are evaluated once when the function is defined, not each time the function is called.
- You should never define default arguments of mutable type unless you know what you are doing.

```
In [18]: def add_to(element, target=None):
             if target is None:
                 target = []
             target.append(element)
             return target
         # Now whenever you call the function without the target argument, a new list is creat
```

0.10 slots Magic

- By default Python uses a dict to store an object's instance attributes.
- The dict wastes a lot of RAM.
- The usage of **slots** to tell Python not to use a dict, and only allocate space for a fixed set of attributes.

```
In [20]: # Without __slots__:

         class MyClass(object):
             def __init__(self, name, identifier):
                 self.name = name
                 self.identifier = identifier
                 self.set_up()
             # ...
```

```
In [21]: # With __slots__:

         class MyClass(object):
             __slots__ = ['name', 'identifier']
             def __init__(self, name, identifier):
                 self.name = name
                 self.identifier = identifier
                 self.set_up()
             # ...
         # The second piece of code will reduce the burden on your RAM. Some people have seen
         # reduction in RAM usage by using this technique.
```

0.11 Virtual Environment

- `pip install virtualenv`
- `virtualenv myproject`
- `source myproject/bin/activate`
- If you want your virtualenv to have access to your systems site-packages, use the –system-site-packages switch when creating your virtualenv like this: `virtualenv --system-site-packages mycoolproject`
- You can turn off the env by typing:
- `deactivate`
- You can use smartcd which is a library for bash and zsh and allows you to alter your bash (or zsh)

0.12 Collections

defaultdict Unlike dict, with defaultdict you do not need to check whether a key is present or not.

```
In [25]: from collections import defaultdict

         colours = (
             ('a', 'Yellow'),
             ('b', 'Blue'),
             ('a', 'Green'),
             ('c', 'Black'),
             ('d', 'Red'),
             ('e', 'Silver'),
         )

         favourite_colours = defaultdict(list)

         for name, colour in colours:
             favourite_colours[name].append(colour)

         print(favourite_colours)
```

```
defaultdict(<class 'list'>, {'a': ['Yellow', 'Green'], 'b': ['Blue'], 'c': ['Black'], 'd': ['R
```

```
In [26]: some_dict = {}
         some_dict['colours']['favourite'] = "yellow"
```

```
        ---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call last)

        <ipython-input-26-6221372f06f1> in <module>()
          1 some_dict = {}
```

```
    ----> 2 some_dict['colours']['favourite'] = "yellow"


        KeyError: 'colours'
```

```
In [28]: import collections
         tree = lambda: collections.defaultdict(tree)
         some_dict = tree()
         some_dict['colours']['favourite'] = "yellow"
         import json
         print(json.dumps(some_dict))
```

```
{"colours": {"favourite": "yellow"}}
```

OrderedDict

- OrderedDict keeps its entries sorted as they are initially inserted.
- Overwriting a value of an existing key doesn't change the position of that key.
- Deleting and reinserting an entry moves the key to the end of the dictionary.

```
In [35]: colours =  {"Red" : 198, "Green" : 170, "Blue" : 160}
         for key, value in colours.items():
             print(key, value)
```

```
Red 198
Green 170
Blue 160
```

```
In [32]: from collections import OrderedDict

         colours = OrderedDict([("Red", 198), ("Green", 170), ("Blue", 160)])
         for key, value in colours.items():
             # Insertion order is preserved
             print(key, value)
```

```
Red 198
Green 170
Blue 160
```

counter Counter allows us to count the occurrences of a particular item.

```
In [36]: from collections import Counter

         colours = (
             ('a', 'Yellow'),
             ('b', 'Blue'),
             ('c', 'Green'),
             ('b', 'Black'),
             ('a', 'Red'),
             ('d', 'Silver'),
         )

         favs = Counter(name for name, colour in colours)
         rint(favs)
```

```
Counter({'a': 2, 'b': 2, 'c': 1, 'd': 1})
```

count the most common lines in a file

```
with open('filename', 'rb') as f:
    line_count = Counter(f)
print(line_count)
```

deque deque provides you with a double ended queue which means that you can append and delete elements from either side of the queue.

```
In [38]: from collections import deque
         d = deque()
         d.append('1')
         d.append('2')
         d.append('3')

         print(len(d))
         print(d[0])
         print(d[-1])
```

```
3
1
3
```

You can pop values from both sides of the deque:

```
In [39]: d = deque(range(5))
         print(len(d))
         d.popleft()
         d.pop()
         print(d)
```

```
5
deque([1, 2, 3])
```

We can also limit the amount of items a deque can hold.

```
d = deque(maxlen=30)
```

You can also expand the list in any direction with new values:

```
In [41]: d = deque([1,2,3,4,5])
         d.extendleft([0])
         d.extend([6,7,8])
         print(d)

deque([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

namedtuple

- A tuple is basically a immutable list which allows you to store a sequence of values separated by commas.
- They are just like lists but have a few key differences.
- The major one is that unlike lists, you can not reassign an item in a tuple

```
In [43]: man = ('Ali', 30)
         print(man[0])

Ali
```

- With namedtuples you don't have to use integer indexes for accessing members of a tuple.
- Unlike dictionaries they are immutable.

```
In [44]: from collections import namedtuple

         Animal = namedtuple('Animal', 'name age type')
         perry = Animal(name="perry", age=31, type="cat")

         print(perry)
         print(perry.name)

Animal(name='perry', age=31, type='cat')
perry
```

- A named tuple has two required arguments. They are the tuple name and the tuple field_names.
- Namedtuple makes your tuples self-document.
- As you are not bound to use integer indexes to access members of a tuple, it makes it more easy to maintain your code.
- As `namedtuple` instances do not have per-instance dictionaries, they are lightweight and require no more memory than regular tuples. This makes them faster than dictionaries.
- attributes in namedtuples are immutable

```
In [45]: from collections import namedtuple

         Animal = namedtuple('Animal', 'name age type')
         perry = Animal(name="perry", age=31, type="cat")
         perry.age = 42


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-45-4e145e02a442> in <module>()
          3 Animal = namedtuple('Animal', 'name age type')
          4 perry = Animal(name="perry", age=31, type="cat")
    ----> 5 perry.age = 42


        AttributeError: can't set attribute
```

```
In [46]: from collections import namedtuple

         Animal = namedtuple('Animal', 'name age type')
         perry = Animal(name="perry", age=31, type="cat")
         print(perry[0])

perry
```

convert a namedtuple to a dictionary

```
In [47]: from collections import namedtuple

         Animal = namedtuple('Animal', 'name age type')
         perry = Animal(name="Perry", age=31, type="cat")
         print(perry._asdict())

OrderedDict([('name', 'Perry'), ('age', 31), ('type', 'cat')])
```

enum.Enum Enums (enumerated type) are basically a way to organize various things.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type="cat")
print(perry._asdict())
```

What if the user types in Cat because they held the Shift key? Or CAT? Or kitten?

```
In [48]: from collections import namedtuple
         from enum import Enum

         class Species(Enum):
             cat = 1
             dog = 2
             horse = 3
             aardvark = 4
             butterfly = 5
             owl = 6
             platypus = 7
             dragon = 8
             unicorn = 9
             # The list goes on and on...

             # But we don't really care about age, so we can use an alias.
             kitten = 1
             puppy = 2

         Animal = namedtuple('Animal', 'name age type')
         perry = Animal(name="Perry", age=31, type=Species.cat)
         drogon = Animal(name="Drogon", age=4, type=Species.dragon)
         tom = Animal(name="Tom", age=75, type=Species.cat)
         charlie = Animal(name="Charlie", age=2, type=Species.kitten)

         # And now, some tests.
         print(charlie.type == tom.type)
         print(charlie.type)
```

```
True
Species.cat
```

access enumeration members,

```
Species(1)
Species['cat']
Species.cat
```

0.13 Enumerate

It allows us to loop over something and have an automatic counter.

```
for counter, value in enumerate(some_list):
    print(counter, value)
```

The optional argument allows us to tell enumerate from where to start the index.

```
In [50]: my_list = ['apple', 'banana', 'grapes', 'pear']
         for c, value in enumerate(my_list, 1):
             print(c, value)

1 apple
2 banana
3 grapes
4 pear
```

```
In [51]: my_list = ['apple', 'banana', 'grapes', 'pear']
         counter_list = list(enumerate(my_list, 1))
         print(counter_list)

[(1, 'apple'), (2, 'banana'), (3, 'grapes'), (4, 'pear')]
```

0.14 Object introspection

introspection is the ability to determine the type of an object at runtime. It is one of Python's strengths.

```
In [52]: my_list = [1, 2, 3]
         dir(my_list)

Out[52]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__gt__',
          '__hash__',
          '__iadd__',
          '__imul__',
          '__init__',
          '__init_subclass__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__reversed__',
          '__rmul__',
          '__setattr__',
          '__setitem__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'append',
          'clear',
          'copy',
          'count',
          'extend',
          'index',
          'insert',
          'pop',
          'remove',
          'reverse',
          'sort']
```

```
In [53]: print(type(''))
         print(type([]))
         print(type({}))
         print(type(dict))
         print(type(3))
```

```
<class 'str'>
<class 'list'>
<class 'dict'>
<class 'type'>
<class 'int'>
```

```
In [54]: name = "Mani"
         print(id(name))
```

```
140172231843712
```

```
In [55]: import inspect
         print(inspect.getmembers(str))
```

```
[('__add__', <slot wrapper '__add__' of 'str' objects>), ('__class__', <class 'type'>), ('__co
```

0.15 Comprehensions

Comprehensions are constructs that allow sequences to be built from other sequences. Three types of comprehensions are supported in both Python 2 and Python 3:

- list comprehensions
- dictionary comprehensions
- set comprehensions
- generator comprehensions

0.15.1 list comprehensions

```
In [56]: multiples = [i for i in range(30) if i % 3 == 0]
         print(multiples)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
In [57]: squared = []
         for x in range(10):
             squared.append(x**2)
```

```
In [60]: # Using List comprehension
         squared = [x**2 for x in range(10)]
         print(squared)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

0.15.2 dict comprehensions

```
In [62]: mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

         mcase_frequency = {
             k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
             for k in mcase.keys()
         }
         print(mcase_frequency)
```

```
{'a': 17, 'b': 34, 'z': 3}
```

0.15.3 set comprehensions

```
In [63]: squared = {x**2 for x in [1, 1, 2]}
         print(squared)
```

```
{1, 4}
```

0.15.4 generator comprehensions

They are also similar to list comprehensions. The only difference is that they don't allocate memory for the whole list but generate one item at a time, thus more memory effecient.

```
In [64]: multiples_gen = (i for i in range(30) if i % 3 == 0)
         print(multiples_gen)
         for x in multiples_gen:
           print(x)
```

```
<generator object <genexpr> at 0x7f7c641b9eb8>
0
3
6
9
12
15
18
21
24
27
```

0.16 Exceptions

In basic terminology we are aware of try/except clause. The code which can cause an exception to occur is put in the try block and the handling of the exception is implemented in the except block.

```
In [65]: try:
             file = open('test.txt', 'rb')
         except IOError as e:
             print('An IOError occurred. {}'.format(e.args[-1]))
```

```
An IOError occurred. No such file or directory
```

Handling multiple exceptions

```
In [66]: try:
             file = open('test.txt', 'rb')
         except (IOError, EOFError) as e:
             print("An error occurred. {}".format(e.args[-1]))
```

```
An error occurred. No such file or directory
```

```
In [67]: try:
             file = open('test.txt', 'rb')
         except EOFError as e:
             print("An EOF error occurred.")
             raise e
         except IOError as e:
             print("An error occurred.")
             raise e
```

```
An error occurred.



        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-67-9f3c58970d73> in <module>()
          6 except IOError as e:
          7     print("An error occurred.")
    ----> 8     raise e


        <ipython-input-67-9f3c58970d73> in <module>()
          1 try:
    ----> 2     file = open('test.txt', 'rb')
          3 except EOFError as e:
          4     print("An EOF error occurred.")
          5     raise e


        FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
```

```
In [68]: try:
            file = open('test.txt', 'rb')
        except Exception:
            # Some logging if you want
            raise
```

```
        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-68-19e2e440f680> in <module>()
          1 try:
    ----> 2     file = open('test.txt', 'rb')
          3 except Exception:
          4     # Some logging if you want
          5     raise
    FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
```

finally clause will run whether or not an exception occurred.

```
In [69]: try:
             file = open('test.txt', 'rb')
         except IOError as e:
             print('An IOError occurred. {}'.format(e.args[-1]))
         finally:
             print("This would be printed whether or not an exception occurred!")
```

```
An IOError occurred. No such file or directory
This would be printed whether or not an exception occurred!
```

Often times we might want some code to run if no exception occurs. This can easily be achieved by using an else clause.

```
In [70]: try:
             print('I am sure no exception is going to occur!')
         except Exception:
             print('exception')
         else:
             # any code that should only run if no exception occurs in the try,
             # but for which exceptions should NOT be caught
             print('This would only run if no exception occurs. And an error here '
                   'would NOT be caught.')
         finally:
             print('This would be printed in every case.')
```

```
I am sure no exception is going to occur!
This would only run if no exception occurs. And an error here would NOT be caught.
This would be printed in every case.
```

0.17 Classes

0.17.1 Instance & Class variables

- Instance variables are for data which is unique to every object
- Class variables are for data shared between different instances of a class

```
In [71]: class Cal(object):
             # pi is a class variable
             pi = 3.142

             def __init__(self, radius):
                 # self.radius is an instance variable
                 self.radius = radius

             def area(self):
                 return self.pi * (self.radius ** 2)

         a = Cal(32)
         a.area()
         a.pi
         a.pi = 43
         a.pi

         b = Cal(44)
         b.area()
         b.pi
         b.pi = 50
         b.pi
```

```
Out[71]: 50
```

New style classes

- Old base classes do not inherit from anything
- New style base classes inherit from object

```
In [72]: class OldClass():
             def __init__(self):
                 print('I am an old class')

         class NewClass(object):
             def __init__(self):
                 print('I am a jazzy new class')

         old = OldClass()
         new = NewClass()
```

```
I am an old class
I am a jazzy new class
```

- This inheritance from object allows new style classes to utilize some magic.
- A major advantage is that you can employ some useful optimizations like **slots**.
- You can use super() and descriptors and the likes.

Magic Methods

```
In [74]: # Whenever an instance of a class is created its __init__ method is called. For examp

         class GetTest(object):
             def __init__(self):
                 print('Greetings!!')
             def another_method(self):
                 print('I am another method which is not'
                       ' automatically called')

         a = GetTest()
         a.another_method()
```

```
Greetings!!
I am another method which is not automatically called
```

```
In [75]: class GetTest(object):
             def __init__(self, name):
                 print('Greetings!! {0}'.format(name))
             def another_method(self):
                 print('I am another method which is not'
                       ' automatically called')

         a = GetTest('Mani')
         # Try creating an instance without the name arguments
         b = GetTest()
```

```
Greetings!! Mani
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-75-66442a242d48> in <module>()
          8 a = GetTest('Mani')
          9 # Try creating an instance without the name arguments
    ---> 10 b = GetTest()


        TypeError: __init__() missing 1 required positional argument: 'name'
```

```
In [78]: class GetTest(object):
             def __init__(self):
                 self.info = {
                     'name':'Mani',
                     'country':'US',
                     'number':12345812
                 }
```

```
            def __getitem__(self,i):
                return self.info[i]

        foo = GetTest()
        foo['number']

Out[78]: 12345812
```

0.17.2 Lambdas

Lambdas are one line functions. They are also known as anonymous functions in some other languages.

```
In [80]: add = lambda x, y: x + y
         print(add(3, 5))

8


In [81]: a = [(1, 2), (4, 1), (9, 10), (13, -3)]
         a.sort(key=lambda x: x[1])
         print(a)

[(13, -3), (4, 1), (1, 2), (9, 10)]


In [85]: # Parallel sorting of lists
         list1 = [2, 3, 4, 1, 8, 9]
         list2 = [4, 8, 1, 2, 5, 6, 9]

         data = sorted(zip(list1, list2))
         data.sort()
         list1, list2 = map(lambda t: list(t), zip(*data))
         print(list1)
         print(list2)

[1, 2, 3, 4, 8, 9]
[2, 4, 8, 1, 5, 6]


In [86]: list1 = [3,2,4,1,1]
         list2 = ['three', 'two', 'four', 'one', 'one2']
         list1, list2 = zip(*sorted(zip(list1, list2)))
         print(list1)
         print(list2)

(1, 1, 2, 3, 4)
('one', 'one2', 'two', 'three', 'four')
```

```
In [87]: list1 = [3,2,4,1,1]
         list2 = ['three', 'two', 'four', 'one', 'one2']
         list1, list2 = (list(t) for t in zip(*sorted(zip(list1, list2))))
         print(list1)
         print(list2)
```

```
[1, 1, 2, 3, 4]
['one', 'one2', 'two', 'three', 'four']
```

0.18 One-Liners

0.19 Simple Web Server

```
python -m http.server
```

0.20 Pretty Printing

```
In [89]: from pprint import pprint
         my_dict = {'name': 'Ma', 'age': 'undefined', 'personality': 'awesome'}
         print(dir(my_dict))
         pprint(dir(my_dict))
```

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '_
['__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

0.20.1 Profiling a script

`python -m cProfile my_script.py`

0.20.2 CSV to json

python -c "import csv,json;print json.dumps(list(csv.reader(open('csv_file.csv'))))"

0.20.3 List Flattening

```
In [91]: import itertools
         a_list = [[1, 2], [3, 4], [5, 6]]
         print(list(itertools.chain.from_iterable(a_list)))
         print(list(itertools.chain(*a_list)))
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

0.20.4 One-Line Constructors

```
In [92]: class A(object):
             def __init__(self, a, b, c, d, e, f):
                 self.__dict__.update({k: v for k, v in locals().items() if k != 'self'})
```

0.20.5 for/else

```
In [93]: fruits = ['apple', 'banana', 'mango']
         for fruit in fruits:
             print(fruit.capitalize())
```

```
Apple
Banana
Mango
```

```
In [94]: for item in container:
             if search_something(item):
                 # Found it!
                 process(item)
                 break
         else:
             # Didn't find anything..
             not_found_in_container()
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-94-90c09bca6d40> in <module>()
    ----> 1 for item in container:
          2     if search_something(item):
          3         # Found it!
          4         process(item)
          5         break


        NameError: name 'container' is not defined
```

```
In [96]: # factors for numbers between 2 to 10
         for n in range(2, 10):
             for x in range(2, n):
                 if n % x == 0:
                     print( n, 'equals', x, '*', n/x)
                     break
             else:
                 # loop fell through without finding a factor
                 print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
```

0.20.6 Python/C API

https://docs.python.org/3/c-api/intro.html

0.21 open

```
f = open('photo.jpg', 'r+')
jpgdata = f.read()
f.close()
```

Explicitly calling close closes the file handle, but only if the read was successful. If there is any error just after f = open(. . .), f.close() will not be called (depending on the Python interpreter, the file handle may still be returned, but that's another story). To make sure that the file gets closed whether an exception occurs or not, pack it into a with statement:

```
with open('photo.jpg', 'r+') as f:
    jpgdata = f.read()
```

The first argument of open is the filename. The second one (the mode) determines how the file gets opened.

- If you want to read the file, pass in r
- If you want to read and write the file, pass in r+
- If you want to overwrite the file, pass in w
- If you want to append to the file, pass in a

```
import io

with open('photo.jpg', 'rb') as inf:
    jpgdata = inf.read()

if jpgdata.startswith(b'\xff\xd8'):
    text = u'This is a JPEG file (%d bytes long)\n'
else:
    text = u'This is a random file (%d bytes long)\n'

with io.open('summary.txt', 'w', encoding='utf-8') as outf:
    outf.write(text % len(jpgdata))
```

0.22 Coroutines

Coroutines are similar to generators with a few differences. The main differences are:

- generators are data producers
- coroutines are data consumers

```
In [98]: # Generator
         def fib():
             a, b = 0, 1
             while True:
                 yield a
                 a, b = b, a+b
```

if we use yield in the above example, more generally, we get a coroutine. Coroutines consume values which are sent to it.

A very basic example would be a grep alternative in Python:

```
In [ ]: def grep(pattern):
            print("Searching for", pattern)
            while True:
                line = (yield)
                if pattern in line:
                    print(line)
```

Well we have turned it into a coroutine. It does not contain any value initially, instead we supply it values externally. We supply values by using the .send() method.

```
In [ ]: search = grep('coroutine')
        next(search)
        search.send("I love you")
        search.send("Don't you love me?")
        search.send("I love coroutines instead!")
```

- The sent values are accessed by yield.
- Why did we run next()? It is required in order to start the coroutine.
- Just like generators, coroutines do not start the function immediately.
- Instead they run it in response to the **next**() and .send() methods.
- Therefore, you have to run next() so that the execution advances to the yield expression.

```
In [ ]: # We can close a coroutine by calling the .close() method:
        search = grep('coroutine')
        # ...
        search.close()
```

0.23 Function caching

- Function caching allows us to cache the return values of a function depending on the arguments.
- It can save time when an I/O bound function is periodically called with the same arguments
- In Python 3.2+ there is an lru_cache decorator which allows us to quickly cache and uncache the return values of a function.

```
In [ ]: from functools import lru_cache

        @lru_cache(maxsize=32)
        def fib(n):
            if n < 2:
                return n
            return fib(n-1) + fib(n-2)

        print([fib(n) for n in range(10)])
```

- The maxsize argument tells lru_cache about how many recent return values to cache.

- We can easily uncache the return values as well by using:

```
fib.cache_clear()
```

0.24 Context Managers

Context managers allow you to allocate and release resources precisely when you want to.

```
In [ ]: with open('some_file', 'w') as opened_file:
            opened_file.write('Hola!')
```

This is equivalent to:

```
In [ ]: file = open('some_file', 'w')
        try:
            file.write('Hola!')
        finally:
            file.close()
```

The main advantage of using a with statement is that it makes sure our file is closed without paying attention to how the nested block exits.

0.24.1 Implementing a Context Manager as a Class

At the very least a context manager has an **enter** and **exit** method defined.

```
In [ ]: class File(object):
            def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
            def __enter__(self):
                return self.file_obj
            def __exit__(self, type, value, traceback):
                self.file_obj.close()
```

```
In [ ]: with File('demo.txt', 'w') as opened_file:
            opened_file.write('Hola!')
```

Our **exit** method accepts three arguments. They are required by every **exit** method.

- The with statement stores the **exit** method of the File class.
- It calls the **enter** method of the File class.
- The **enter** method opens the file and returns it.
- The opened file handle is passed to opened_file.
- We write to the file using .write().
- The with statement calls the stored **exit** method.
- The **exit** method closes the file.

0.24.2 Handling Exceptions

```
In [ ]: with File('demo.txt', 'w') as opened_file:
            opened_file.undefined_function('Hola!')
```

Let's list the steps which are taken by the with statement when an error is encountered:

- It passes the type, value and traceback of the error to the **exit** method.
- It allows the **exit** method to handle the exception.
- If **exit** returns True then the exception was gracefully handled.
- If anything other than True is returned by the **exit** method then the exception is raised by the with statement.

```
In [ ]: # Lets try handling the exception in the __exit__ method:

        class File(object):
            def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
            def __enter__(self):
                return self.file_obj
            def __exit__(self, type, value, traceback):
                print("Exception has been handled")
                self.file_obj.close()
                return True

        with File('demo.txt', 'w') as opened_file:
            opened_file.undefined_function()
```

Implementing a Context Manager as a Generator
We can also implement Context Managers using decorators and generators. Python has a contextlib module for this very purpose. Instead of a class, we can implement a Context Manager using a generator function.

```
In [ ]: from contextlib import contextmanager

        @contextmanager
        def open_file(name):
            f = open(name, 'w')
            yield f
            f.close()
```

Let's dissect this method a little.

- Python encounters the yield keyword. Due to this it creates a generator instead of a normal function.
- Due to the decoration, contextmanager is called with the function name (open_file) as it's argument.
- The contextmanager decorator returns the generator wrapped by the GeneratorContextManager object.

- The GeneratorContextManager is assigned to the open_file function. Therefore, when we later call the open_file function, we are actually calling the GeneratorContextManager object.

```
In [ ]: # So now that we know all this, we can use the newly generated Context Manager like th

        with open_file('some_file') as f:
            f.write('hola!')
```