Introduction

- Knowing each other
- Expectations

Setup

- Before stepping into the world of Python lets setup.
- We will use Python 3.5.3 and Django 1.10 (More on Django later)
- Download Python and editor PyCharm (windows and mac)
- https://www.python.org/downloads/
- https://www.jetbrains.com/pycharm/download
- Install Python (On windows, select option “add Python to PATH”) and PyCharm

Setup…

- Set PATH variable. In windows, if you set PATH at installation time, then you are good.
- Open terminal and test the installation
  - Windows: $py
  - Mac: type $python

Setup: Python package index

- Package repository for python (https://pypi.python.org/pypi)
- Python package manager: pip & pip3

  eg: pip3 install Django==1.10.6
- Package manager is included with python distribution.
- Type ‘pip –version’ to test on both mac and windows

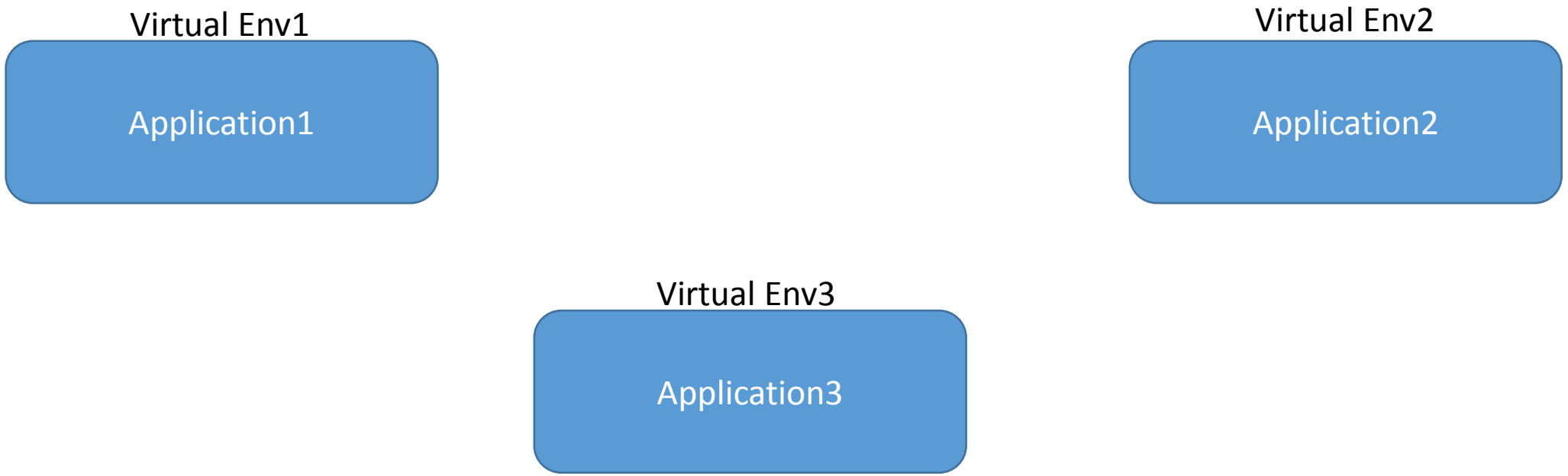
Setup: Virtual Environment

- What is virtual environment? More on this later, first lets setup this.
- Mac:
  - Go to workspace
  - $ **pip3 install virtualenv**
  - $ **virtualenv <env>** (folder will be created with name ‘env’)
  - $ **source <env>/bin/activate**
  - Go inside env and create folder src
  - Your source files will live in src
- Windows:
  - You can use Pycharm IDE to create while creating new project OR
  - Go to workspace
  - **$ pip3 install virtualenv**
  - **$ virtualenv <env>** (folder will be created with name ‘env’)
  - **$ <env>/Scripts/activate**
  - Go inside env and create folder src
  - Your source files will live in src

Virtual Environment

- What is virtual environment?
- Allow python packages to be installed in isolated location for a particular application
- User permissions to upgrade global packages
- No package upgrade issues across apps

Virtual Environment...



Hello Python

- Create file hello.py in src folder

```
print("Hello, World")
```

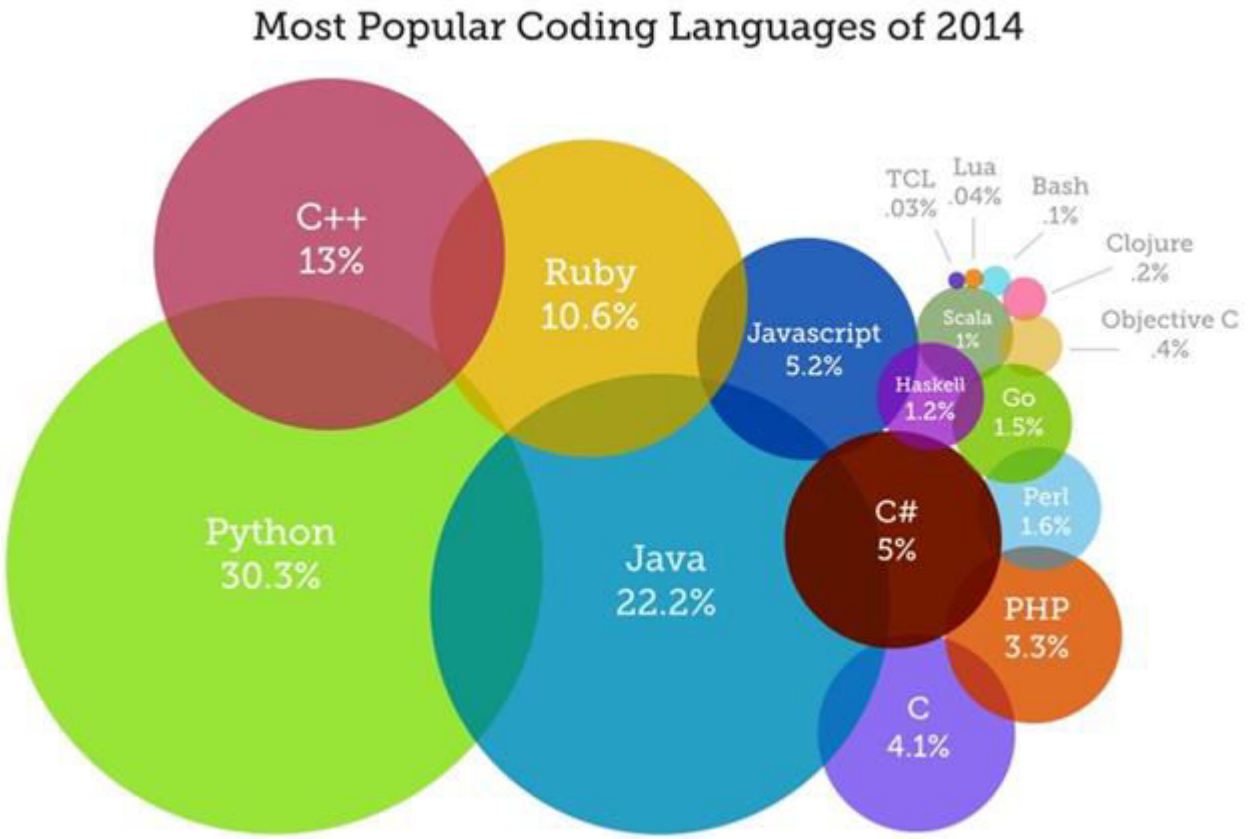
- Run it

```
python3 hello.py
```

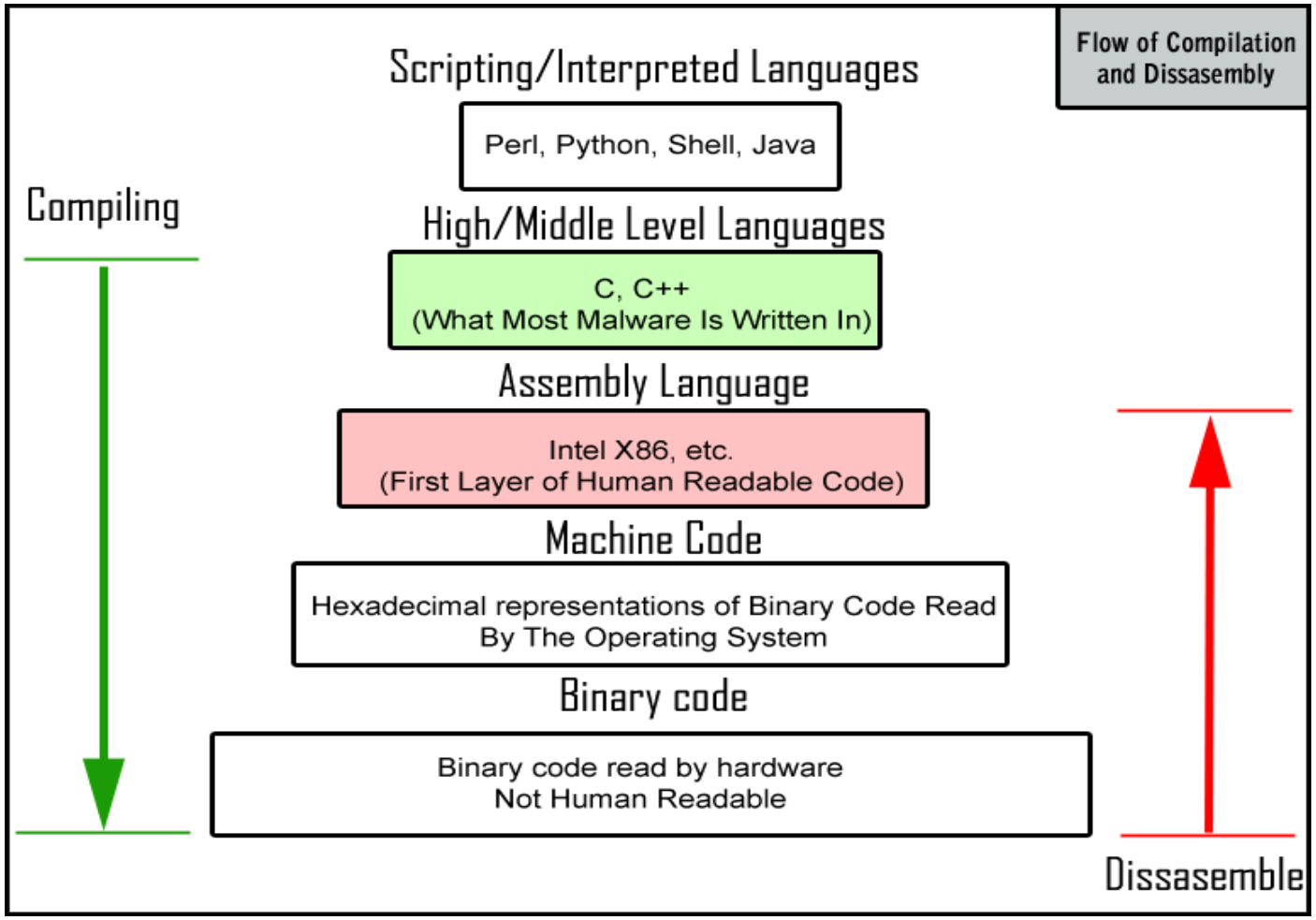
Python…

- What is Python?
- Why Python?

Popularity in 2014



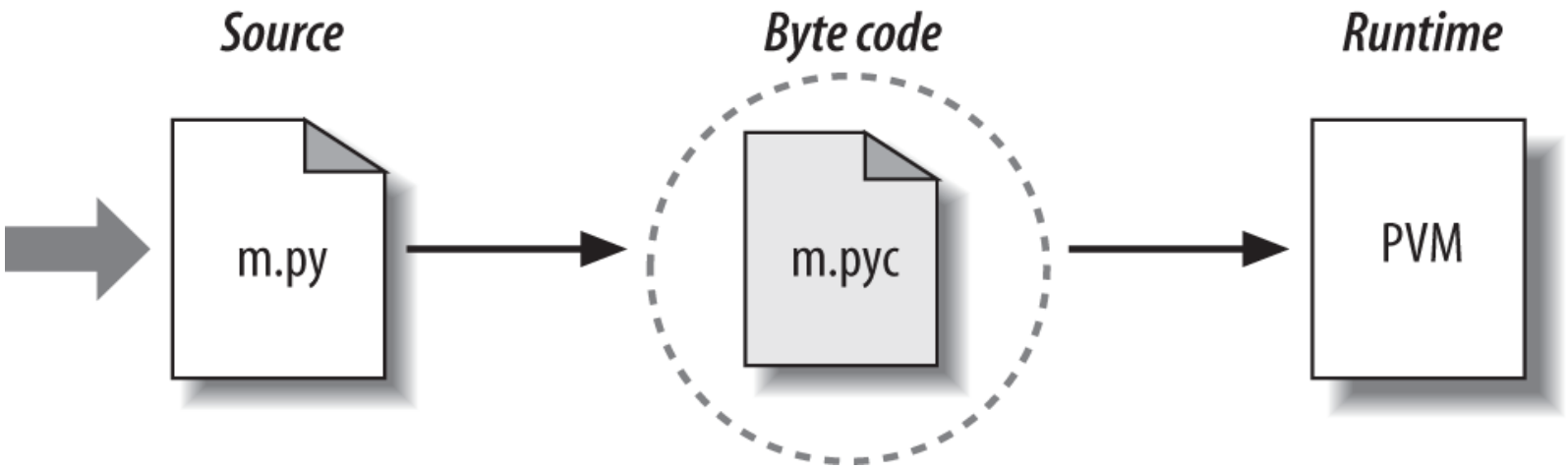
High level language



Python…

- Scriptable
- Object Oriented
- Interpreted
- Dynamic
- Extensible
- Versions (2.x and 3.x)
- General purpose (Wide variety of tasks)
- Structured

Interpreted



A bit confusing stuff ☺. Interpreted but then what is this byte code thing.

Dynamically typed

- Java

  String value;
  value = “java”
  value = 5 (not allowed, will fail at compile time)

- Python

  a = 4.5

  a = “python”

  a = 4

A not so convincing example ☺

Structured

- Indentation in Python is mandatory. We will see this while coding.
- Code readability

Hands-On

- Time to get out of presentation mode and wear the programmer's hat

```
"""
Multi-line comment
"""

# Single line comment

print("hello, world")

'''
We can define strings using ' (single quotes) or using " (double quotes)
Same goes for comments, did you notice this one.
'''
```

Hands-On

```
# Working with variables is damn easy
an_int = 1
a_string = "We won't work with other types today. Yes, there are many more."

'''
There is no verbosity like – int anInt = 1; or String aString = "Something";
'''
```

Hands-On

```
# Programming is all about decision making, is not it?
if an_int == 1:
    print(a_string)

# A decision without a negative case is not so useful
if an_int == 2:
    print(a_string)
else:
    print("Damn it was not true!!!")

# Ah! that was nice but how can I take more than one decisions
if an_int == 2:
    print("It is 2 indeed")
elif an_int == 1:
    print("It is 1 indeed")
else:
    print("I seriously have not idea, what it is")
```

Hands-On

```
'''
Do we just keep scripting in Python or can we package snippets and reuse
Did not you realize, what print is? Yes, it is a function.

A callable, reusable and self contained unit of code. Provides a logical grouping and
helps in organizing snippets to perform unit of work.

Disclaimer: I am NOT good at definitions and this one is purely self cooked :-)
'''
```

Hands-On

```
def greet_awesome_people():
    print("Hello Awesome People.I am a dumb function but teaches a very powerful thing. \nGuess what?")


# Guess what?
greet_awesome_people()

# Same goes for me, guess guess :-)
def i_am_bit_smarter(message):
    print(message)


# And same goes for me
def i_am_bit_more_smarter(a, b):
    return a + b

i_am_bit_smarter("Custom Message>> Sum of 10 and 2 is : " + str(i_am_bit_more_smarter(10, 2)))
```

Hands-On

```
'''
Assignment
----------

Write the smartest calculator which:
- Works only with integer
- Handles add, subtract, mul and divide

client should be able to use your calculator like:
add(10,2), subtract(11, 3) etc.
'''
```

Variables

- Define variable with name and value separated by = sign
- Data type is not required
- No way to declare a variable without assigning it an initial value

```
>>> a = 5
>>> b = 4.5
>>> c = True
```

- Python interpreter in action
- What's the type of variables?

```
>>> print(type(a))
>>> print(type(b))
>>> print(type(c))
```

Variables…

- Assign multiple values to multiple variables in one line

  >>> a, b, c = 1, 2, 3

- Number should match
- Single value to several variables

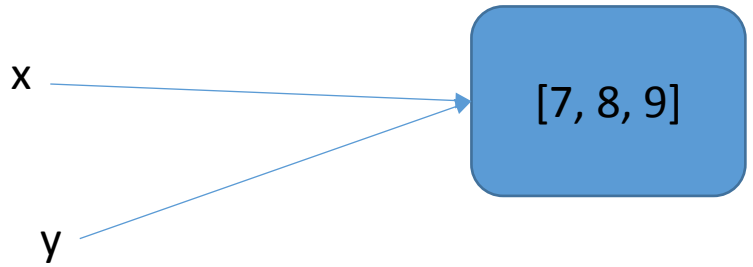
  >>> a = b = c = 1

  a, b, and c, are all independent

  changing a will not affect b and c

Changing Variable Value

- Variables are references
  >>> x = y = [7, 8, 9]



- What happens when you say x[0] = 22 ?
  >>> print(y)
- What happens when you say
  >>> x = [7, 8, 9]
- More on this later ☺

Datatypes

- Booleans (True, False)

  bool type is subclass of int

- Numbers (int, float, complex)

  b = 100, b = 100.5, b=2 + 1j

- Sequences and collections

  str, tuple, list, set, dict

Booleans

- Boolean is subclass of int
- True and False are it’s instances

```
>>> issubclass(bool, int) = ??
>>> isinstance(True, bool) = ??
>>> True + False = ??
>>> int(True) = ??
```

List

- Square brackets
- Ordered collection of objects
- Mutable

```
>>> int_list = [1, 2, 3]
>>> string_list = ['abc', 'defghi']
>>> b = ['a', 1, 'python', (1, 2), [1, 2]]  … why?
>>> nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

List...

- Index access

  >>> b = ['a', 1, 'python', (1, 2), [1, 2]]

  >>> print(b[0])
- Change value at any index

  >>> b[1] = 0
- Append

  >>> b.append(10)
- İnsert before index

  >>> b.insert(1, 'python2')
- Remove the first occurrence of value

  >>> b.remove('python2')

List...

- index of the first item whose value is x
  >>> b.index('a')
- Length
  >>> len(b)
- Count occurrence of any item in list
  >>> b.count('a')
- Reverse the list
  >>> b.reverse()
- Remove and return item at index (default last)
  >>> b.pop(0)

List…

- Iterating over list

```
l = [1, 'abc', 'c']
for element in l:
    print(element)
```

Tuples

- Ordered collections of objects
- Parenthesis
- Fixed length
- immutable

```
>>> ip_address = ('10.0.90.21', 8080)
>>> b = ('a', 1, 'python', (1, 2))
>>> b[2] = ??
>>> b[2] = 5   (what happens?)
```

Set

- Curly braces
- Unordered collection of unique objects

```
>>> a = {1, 2, 'a'}
```

```
l = [1, 'abc', 'c']
for element in l:
    print(element)


s = set(l)
for element in s:
    print(element)
```

Dictionary

- Key value pair

  >>> a = {1: 'one', 2: 'two'}

  >>> a[1] == ??

Dictionary…

```
my_map = {
    'one' : 1,
    'two' : 2,
    'three' : 3
}


for key in my_map.keys():
    print('key = {}  and value = {}'.format(key, my_map[key]))


for key, value in my_map.items():
    print(key, value)
```

Scope

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1


def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
# print(c)
# print(d)
```

Assignment

Write a file listUtils.py which has following apis:

- Find last index of element, return None if element not found (optimized)

  def find_last_index(list, element)

- Find if given element is present inside list or not. Return True if present else return false.

  def is_present(list, element)

- Given list of integers, find how many time each integer is present in list. Print number and its count

  def count_and_print_numbers(list, element)

Recap

- Variables
- Collections
- Scope

Scope…

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1


def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
# print(c)
# print(d)
```

Scope…

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Scope…

```
a = 0
def my_function():
    print(a)
my_function()


def my_function1():
    a = 3
    print(a)

my_function1()

print(a)


def my_function2():
    global a
    a = 3
    print(a)

my_function2()

print(a)
```

Scope…

- *Assignment operator always creates variable in local scope*

```
# how about passing an argument
def greet_again(message):
    print(message)

    if message == "abc":
        c = 1
    return c  # note c is accessible outside if
```

Scope conclusion

- Innermost scope, which is searched first, contains the local names
- Scopes of any enclosing functions
- Next-to-last scope contains the current module’s global names
- Outermost scope (searched last) is the namespace containing built-in names

Mutable vs Immutable

- Common immutable type:

1. numbers: int(), float(), complex()
2. immutable sequences: str(), tuple()

- Common mutable type

1. mutable sequences: list()
2. set type: set()
3. mapping type: dict()

Mutable vs Immutable…

- Keep memory in mind.
- If you can change value at memory location → mutable

```
>>> a = 5
>>> id(a) ??
>>> a = 8
>>> id(a) ??
```

Mutable vs Immutable…

```
>>> list1=[1,2,3]
>>> id(list1) ??
>>> list1 +=  [4]
>>> id(list1) ??

>>> t = ('some', 'tuple')
>>> id(t) ??
>>> t += ('some', 'other', 'tuple')
>>> id(t) ??
```

User input

- name = input("What is your name? ")

Function

- The keyword ’def’

  def function_name(parameters):

  statement(s)

- *function_name* is known as the *identifier* of the function
- *parameters* is an optional list of identifiers
- *statement(s) – function body* – nonempty sequence of statements

Function…

```
# Let play with functions again :)
def greet_again():
    print("Hello Again")


# how about passing an argument
def greet_again(message):
    print(message)


# what is argument type?
def greet_again_type(message):
    print(type(message))
    print(message)
```

Function…

```
# no return type required, what does that mean?
def multiple_types(x):
    if x < 0:
        return "Return String!"
    else:
        return 0

print(multiple_types(1))
print(multiple_types(-1))
```

Function…

```
# how to pass variable arguments
# Arbitrary number of positional arguments
def var_arg_function(*args): # args will be a tuple containing all values that are passed in
    for i in args:
        print("printing", i)


var_arg_function(1, 2, 3)


# If you have an array, expand it using *
a = [1, 2, 3, 5]
var_arg_function(*a)
```

Function…

```
# Arbitrary number of keyword arguments
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)


func(value1=1, value2=2, value3=3)
func()
my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)
```

Modules - Intro

- File containing definitions and statements
- Python file
- Modules can be imported
- Import a function from module
- Use alias for modules

Modules…

- Create file fibo.py

```
def fib(n):    # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
#
# print(fib2(10))
#
print(fib(10))
```

Create file module1.py and use fibo.py

```
from day3 import fibo
print(fibo.fib(10))
```

```
import day3.fibo as h
print(h.fib(10))
```

```
import day3.fibo
print(day3.fibo.fib(10))
```

Class

- Python is object oriented language
- Classes: data + methods to operate on data

Class…

```
"""A class with an instance variable."""
class MyClass(object):
    def __init__(self, value):
        self.instance_var = value

# create an object
obj = MyClass("It is an instance variable")

# access an object data members
print(obj.instance_var)
```

```
print("\n\n-------- Class With Methods ----------")

class MyClassWithMethod(object):
    def __init__(self, value):
        self.instance_var = value

    def my_method(self):
        return self.instance_var

obj = MyClassWithMethod(1)

# That is how we access method
print(obj.my_method())
```

Class…

```
print("\n\n-------- Class With Class Member -----------")

# class with class members
class MyClassWithMembers:
    class_var = 2

# Access class member with class name directly
print(MyClassWithMembers.class_var)

obj1 = MyClassWithMembers()
obj2 = MyClassWithMembers()

# Access class member with objects
print(obj1.class_var)
print(obj2.class_var)
```

```
# Setting value for class member with class
MyClassWithMembers.class_var = 3

# value stayed same across
print("value stayed same across")
print(MyClassWithMembers.class_var)
print(obj1.class_var)
print(obj2.class_var)


# Setting value for class member via object
obj1.class_var = 4

print(MyClassWithMembers.class_var)

'''
value for obj1 has changed, obj1 did not have class_var
member and thus it got added as member for obj1.
'''
print(obj1.class_var)
print(obj2.class_var)
```

Class - observations

- Every Python class must have an initializer
- Class attributes
- Instance attributes. Should be inside __init__() method
- The word **self** as first parameter for all functions
- private vs public elements?
- User dot(.) operator to access attributes and call functions.

Class - observations

- obj.function(arg1) is equivalent to ClassName.function(obj, arg1)
- Eg:

```
class A(object):
    def f(self, x):
        return 2 * x

a = A()
print(A.f(a,2))
print(a.f(2))
```

Shadowing class variables

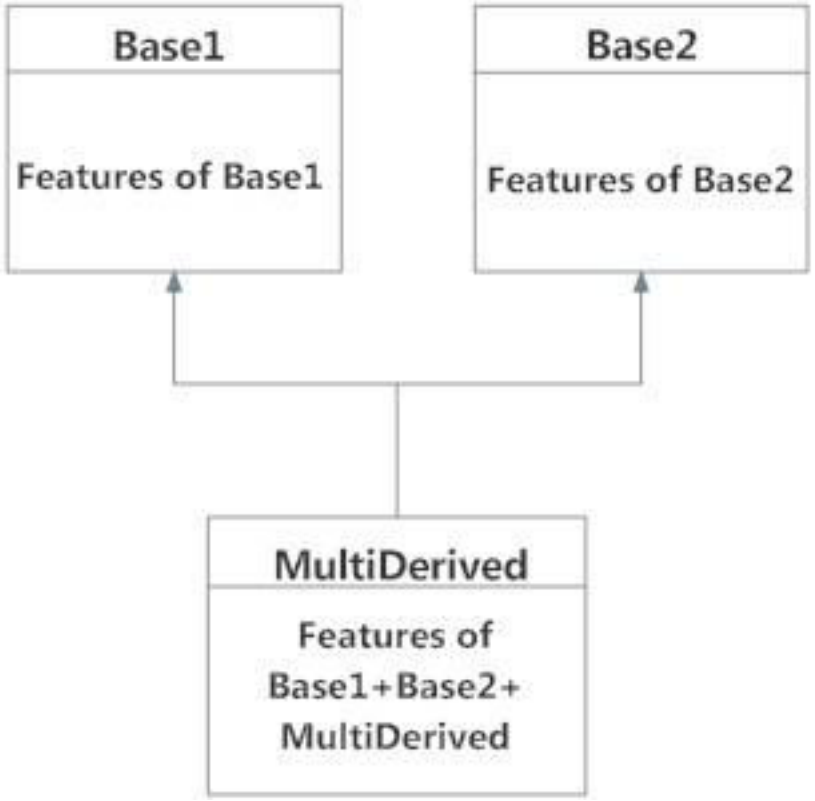
```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

print(C.x)
print(C.y) # AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
print(c1.x)
print(c1.y)

c2 = C(4)
print(c2.x) # 2
print(c2.y) # 4

#shadowing a class variable

c2.x = 4
print(c2.x) # 4
print(C.x) #2
```

Inheritance - introduction

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)


class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super().__init__(s, s)
        self.s = s

s = Square(4)
print(s.area())
```

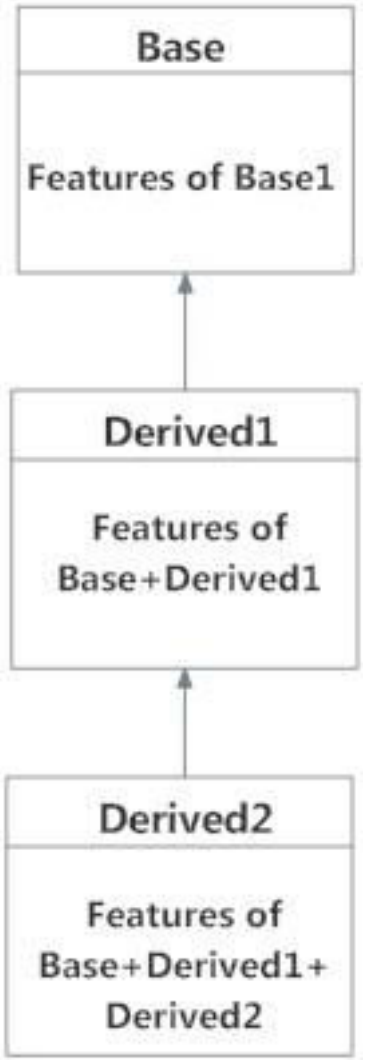
Inheritance…

- Class can inherit from any other class



Inheritance…

- Multi level inheritance is allowed



Method overriding

- Override method of super class in sub class
- If method is not found in subclass, it is searched in base class
- Object class comes at top of hierarchy.

Method overriding

```
class Base():
    def m1(self):
        print("in base m1")


class Sub(Base):
    def m2(self):
        print("in Sub m2")


s = Sub()
s.m1()
s.m2()
```

Multiple Inheritance

- Class can extend from multiple base classes
- The diamond problem in python?

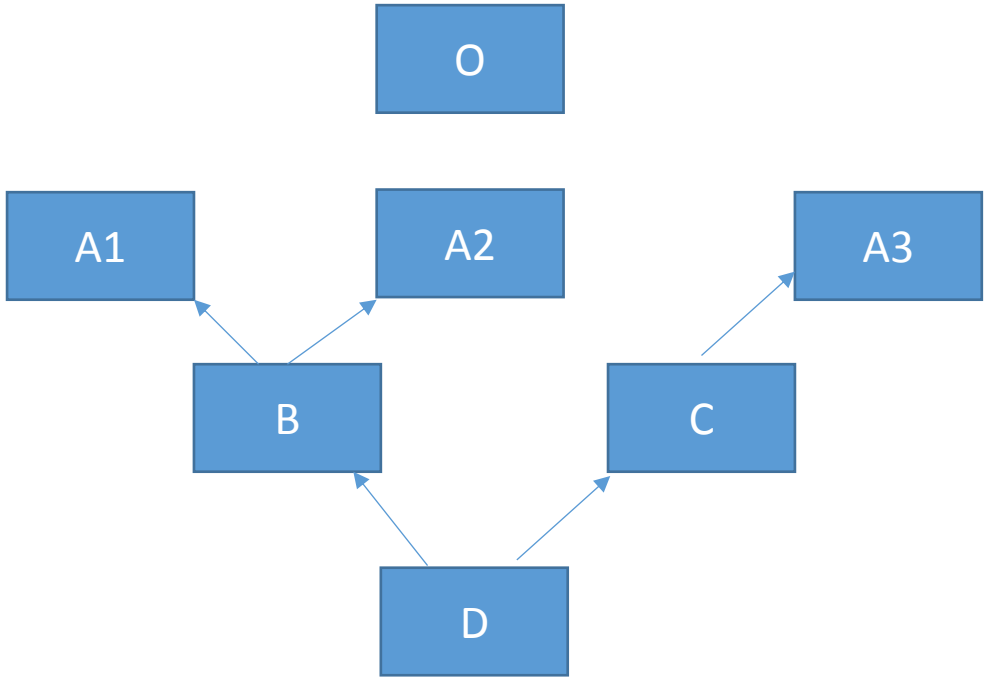
Multiple Inheritance…

- Where fb.foo will be taken in below example? from Foo or from Bar?

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar'
    bar = 'attr bar of Bar'


class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'


fb = FooBar()
print(fb.foo)
```

Method Resolution Order

- Depth first (e.g. FooBar then Foo)
- Left to Right



D B A1 A2 C A3 O

Method overloading

- Methods are objects
- Overloading is done using one function

```
def func1(a, b=None):
    print(a, b)

func1(1, 2)
func1(3)
```

Assignment

Write a file bst.py which has following apis:

- Implement a binary search tree where every node has value and left and right child. Use class to represent node.
- Implement sorting of elements using created BST, given there are n element in BST
- In you method your input should be list of unsorted elements and output should be sorted list, method should create BST from input list and sort the elements using this BST and return sorted list.

Recap

- Functions
- Classes

Class…

```
"""A class with an instance variable."""
class MyClass(object):
    def __init__(self, value):
        self.instance_var = value

# create an object
obj = MyClass("It is an instance variable")

# access an object data members
print(obj.instance_var)
```

```
print("\n\n-------- Class With Methods ----------")

class MyClassWithMethod(object):
    def __init__(self, value):
        self.instance_var = value

    def my_method(self):
        return self.instance_var

obj = MyClassWithMethod(1)

# That is how we access method
print(obj.my_method())
```

Class...

```
print("\n\n-------- Class With Class Member -----------")

# class with class members
class MyClassWithMembers:
    class_var = 2

# Access class member with class name directly
print(MyClassWithMembers.class_var)

obj1 = MyClassWithMembers()
obj2 = MyClassWithMembers()

# Access class member with objects
print(obj1.class_var)
print(obj2.class_var)
```

```
# Setting value for class member with class
MyClassWithMembers.class_var = 3

# value stayed same across
print("value stayed same across")
print(MyClassWithMembers.class_var)
print(obj1.class_var)
print(obj2.class_var)


# Setting value for class member via object
obj1.class_var = 4

print(MyClassWithMembers.class_var)

'''
value for obj1 has changed, obj1 did not have class_var
member and thus it got added as member for obj1.
'''
print(obj1.class_var)
print(obj2.class_var)
```

Class - observations

- Every Python class must have an initializer
- Class attributes
- Instance attributes. Should be inside __init__() method
- The word **self** as first parameter for all functions
- private vs public elements?
- Use dot(.) operator to access attributes and call functions.

Class - observations

- obj.function(arg1) is equivalent to ClassName.function(obj, arg1)
- Eg:

```
class A(object):
    def f(self, x):
        return 2 * x

a = A()
print(A.f(a,2))
print(a.f(2))
```

Shadowing class variables

```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

print(C.x)
print(C.y) # AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
print(c1.x)
print(c1.y)

c2 = C(4)
print(c2.x) # 2
print(c2.y) # 4

#shadowing a class variable

c2.x = 4
print(c2.x) # 4
print(C.x) #2
```

Shadowing…

- What will be the output of this code?

```
class Dog:

    tricks = []

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
```

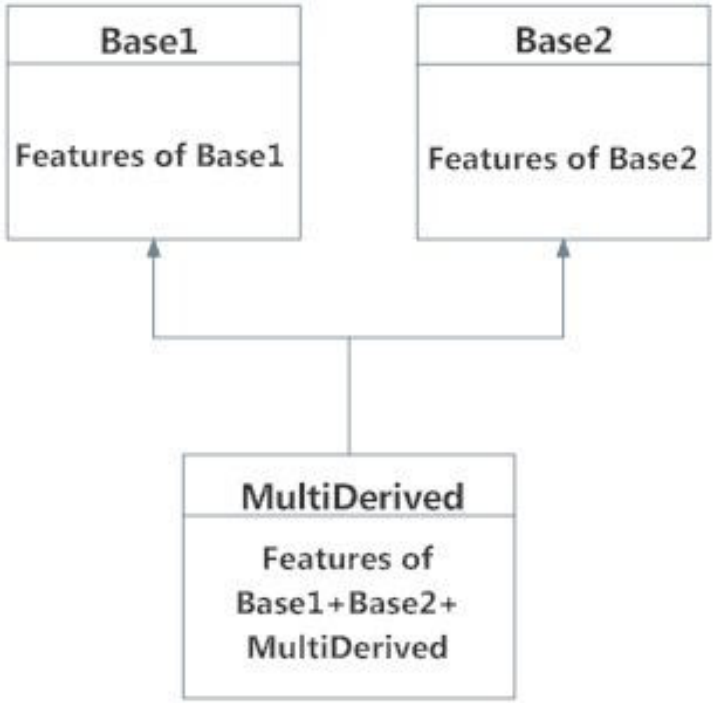
Shadowing...

- Correct use will be to use instance attribute
- Each dog object has its own list of tricks

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
print(e.tricks)
```

Inheritance

- Class can inherit from any other class

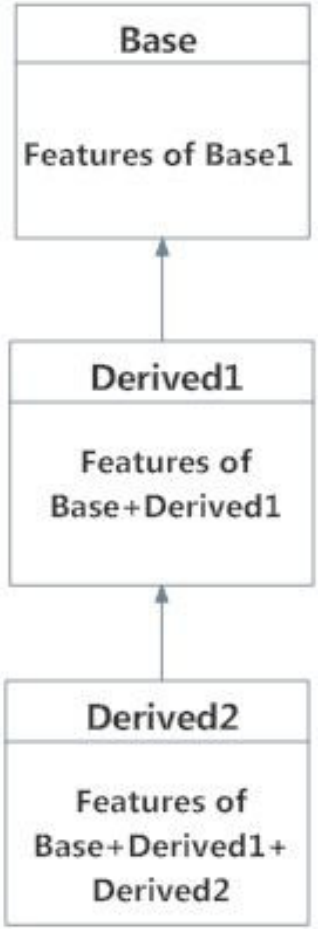


Inheritance…

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)


class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super().__init__(s, s)
        self.s = s

s = Square(4)
print(s.area())
```

Inheritance…

- Multi level inheritance is allowed



Method overriding

- Override method of super class in sub class
- If method is not found in subclass, it is searched in base class
- Object class comes at top of hierarchy.

Method overriding

```
class Base():
    def m1(self):
        print("in base m1")


class Sub(Base):
    def m2(self):
        print("in Sub m2")


s = Sub()
s.m1()
s.m2()
```

Example

```
class Pet(object):
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def get_name(self):
        return self.name

    def get_species(self):
        return self.species

    def __str__(self):
        return "%s is a %s" % (self.name, self.species)
```

Example

```
class Dog(Pet):
    def __init__(self, name, chases_cats):
        self.chases_cats = chases_cats
        super().__init__(name, "Dog")
        #Pet.__init__(self, name, "Dog")

    def chases_cats(self):
        return self.chases_cats

    def get_name(self):
        return "OVERRIDDEN NAME"
```

```
class Cat(Pet):
    def __init__(self, name, hates_dogs):
        Pet.__init__(self, name, "Cat")
        self.hates_dogs = hates_dogs

    def hates_dogs(self):
        return self.hates_dogs
```

Example

- How to call super constructor
- Place where super is called?
- How overriding works?
- The isinstance and issubclass methods (isA relationship)

```
dog1 = Dog("dog1", "chases cat")
dog2 = Dog("dog2", "chases cat as well")
print(dog1.get_name())
print(dog2.get_name())
print(dog1.__str__())
```

- print(Pet.get_name(dog1)) – what happens here?

Multiple Inheritance

- Class can extend from multiple base classes
- The diamond problem in python?

Multiple Inheritance…

- Where fb.foo will be taken in below example? from Foo or from Bar?

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar'
    bar = 'attr bar of Bar'


class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'


fb = FooBar()
print(fb.foo)
```

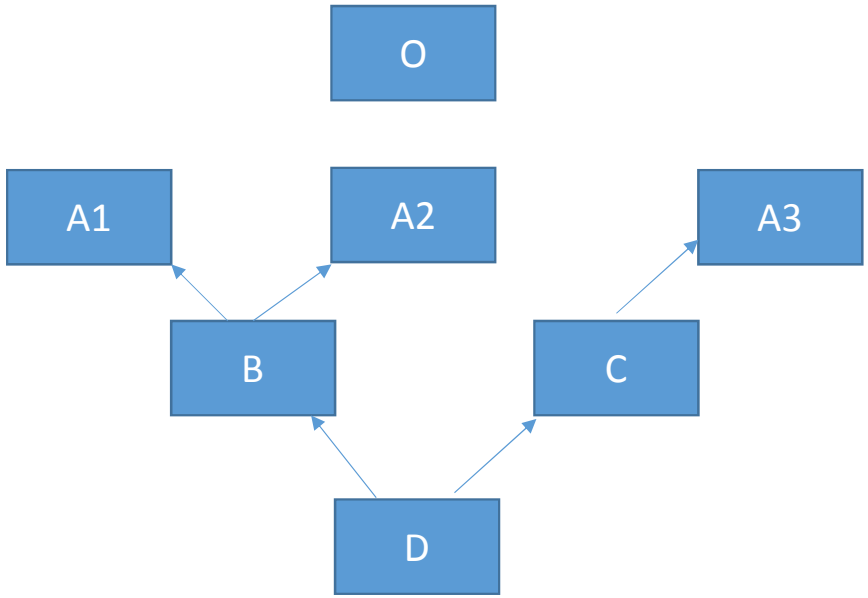
Multiple Inheritance…

- What if Foo, Bar order is reversed

```
class FooBar(Bar, Foo):
    foobar1 = "foobar1"

fb1 = FooBar
print(fb1.foo)
```

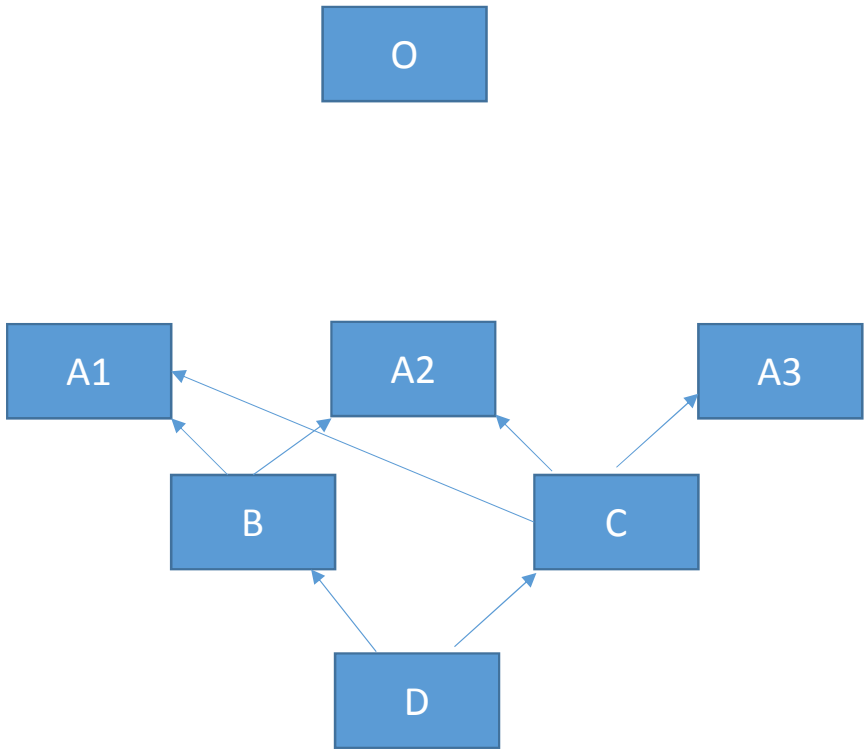
Method Resolution Order

- Depth first (e.g. FooBar then Foo)
- Left to Right



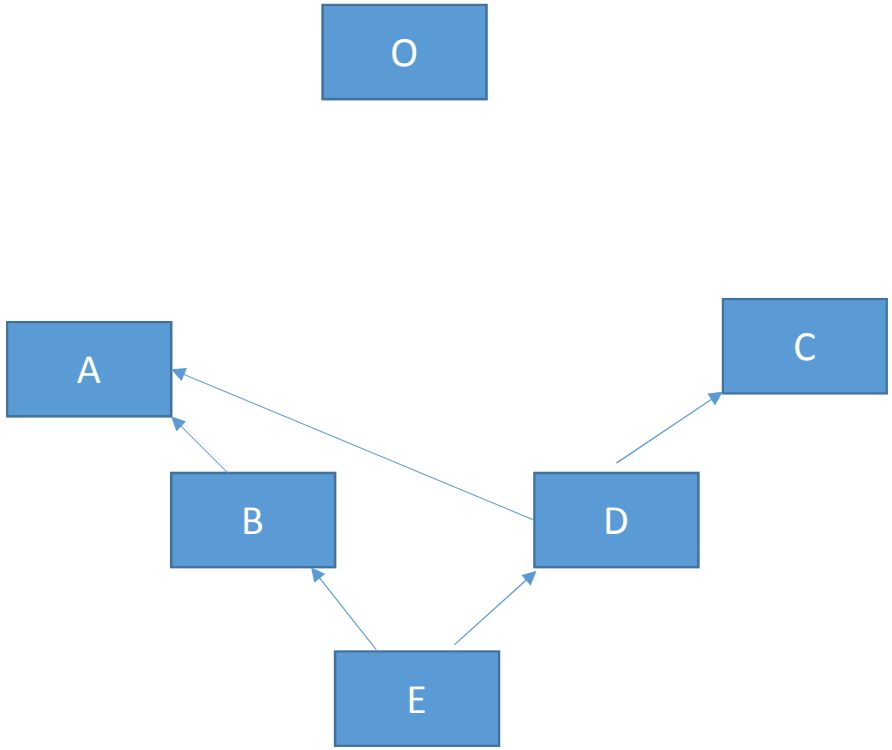
D B A1 A2 C A3 O

Method Resolution Order



D B C A1 A2 A3 O

Method Resolution Order



E B D A C O

Method overloading

- What is method overloading

Method overloading

- No method overloading in python
- One function with Mix and match parameters

```
def func1(arg1, arg2=None):
    if arg2 is not None:
        print(arg1, arg2)
    else:
        print(arg1)

func1(1, 2)
```

```
def func2(arg1, *args):
    if args:
        for item in args:
            print(item)
    else:
        print(arg1)

func2(1, 2, 3, 4, 5)
```

```
def func3(arg1, **args):
    if args:
        for k, v in args.items():
            print(k, v)
    else:
        print(arg1)

func3(1, arg2=2, arg3=3)
func3(1)
func3(3, arg2=2, arg3=3)
```

Method overloading

- Mix and match parameters

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

Lambda

- Inline anonymous inner function
- contains single expression
- Value after : is returned
- Lambda is a tool for building functions

```
def greeting():
    return "Hello"

print(greeting())


greet = lambda : "Hello"

print(greet())
```

Lambda

- Can take parameters
- Can take arbitrary arguments

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

Lambda

- Mostly used in transformations
- Code is clean

```
l = [1, -2, 3, -4, 5, 7]
l = list(filter(lambda x: x>0, l))
print(l)

my_list = [1, -2, 3, -4, 5, 7]
print(my_list)
l = list(map(lambda x: abs(x), my_list))
print(l)
```

Lambda: cleaner code

```
1   def __init__(self, parent):
2       """Constructor"""
3       frame = tk.Frame(parent)
4       frame.pack()
5
6       btn22 = tk.Button(frame,
7           text="22", command=self.buttonCmd22)
8       btn22.pack(side=tk.LEFT)
9
10      btn44 = tk.Button(frame,
11          text="44", command=self.buttonCmd44)
12      btn44.pack(side=tk.LEFT)
13
14  def buttonCmd22(self):
15      self.printNum(22)
16
17  def buttonCmd44(self):
18      self.printNum(44)
```

it is much easier (and clearer) to code

```
1   def __init__(self, parent):
2       """Constructor"""
3       frame = tk.Frame(parent)
4       frame.pack()
5
6       btn22 = tk.Button(frame,
7           text="22", command=lambda: self.printNum(22))
8       btn22.pack(side=tk.LEFT)
9
10      btn44 = tk.Button(frame,
11          text="44", command=lambda: self.printNum(44))
12      btn44.pack(side=tk.LEFT)
```

Assignment

- You are a bank and you want to open user bank accounts. There can be different type of bank accounts for example salary account, zero balance account, minimum balance account.
- You have to support following operations

1) open_account

2) withdraw

3) deposit.

- Create classes and use inheritance, composition if needed. And implement methods.

Recap

- Multiple Inheritance
- MRO
- Closures
- Lambda

Iterators

- How for loop works
- Iterator object on container
- StopIteration Exception tells for loop to stop

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
```

Iterators…

- The iter() method
- Fourth call in example will throw StopIteration exception
- How will you create you own iterator?

```
s = 'abc'
it = iter(s)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Iterators…

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

rev = Reverse('spam')
iter(rev)
for char in rev:
    print(char)
```

Iterators…

```
print('***** fib *****')
class Fib:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib

for i in Fib(10):
    print(i)
```

Iterators…

Your range function

```
print('**** print 0 to n-1 ****')
class CustomRange:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.curr = 0
        return self

    def __next__(self):
        numb = self.curr
        if self.curr >= self.max:
            raise StopIteration
        self.curr += 1
        return numb

for i in CustomRange(10):
    print(i)
```

Generators

- Way to create iterators with ease
- Regular functions but use the **yield** statement whenever they want to return data
- Remembers the data values and last executed statement
- Create next and iter functions for you.
- When generators terminate, they automatically raise StopIteration
- Generators are used either by calling the next method on the generator object or using the generator object in a *for* loop

Generator working

```
def foo():
    print("begin")
    for i in range(3):
        print("before yield", i)
        yield i
        print("after yield", i)
        print("end")

f = foo()

print(next(f))
print(next(f))
print(next(f))
#print(next(f))   #uncomment it and see
```

Generators…

```
print('***** Fibonacci without Generator *****')
def fib(max):
    numbers = []
    a, b = 0, 1
    while a < max:
        numbers.append(a)
        a, b = b, a + b
    return numbers

print(fib(10))
```

```
print('***** Fibonacci using Generator *****')
def fib(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a + b

gen = fib(10)
print(gen)
print(next(gen))
```

Generators…

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]


for char in reverse('golf'):
    print(char)
```

Generators…

- Let say at any point you need next fibonacci number than previously generated. What will you do?

```
print('***** Fibonacci with infinite loop *****')
def fib():
    a, b = 0, 1
    while True:
        print(a)
        a, b = b, a + b

fib()
```

Can we achieve same in efficient manner?

Generators…

```
print('***** Fibonacci using Generator *****')
def fib(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a + b

gen = fib(10)
print(gen)
print(next(gen))
```

Last example

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(next(seq))
    except StopIteration:
        pass
    return result

print(take(5, squares()))
```

Lets Decorate

```
def getMessage(version):
    return "Python {0} is awesome".format(version)
```

Given above function we want to get the message and then transform it in MarkDown. Specifically, we want to create heading, bold text and italicize text.

Lets Decorate

```
# heading wrapper
def heading(func):
    def wrapper(message):
        return "# {0}".format(func(message))
    return wrapper

# bold wrapper
def bold(func):
    def wrapper(message):
        return "**{0}**".format(func(message))
    return wrapper

# italics wrapper
def italic(func):
    def wrapper(message):
        return "*{0}*".format(func(message))
    return wrapper
```

Lets Decorate

We decorated our function!!!

```
quote = getMessage
quote("version")
print(heading(bold(italic(quote)))("3"))
```

Lets Decorate

Decorating using annotations

```
# using decorator anotations
@heading
@bold
@italic
def decoratedMessage(version):
    return getMessage(version)

print(decoratedMessage("1"))
```

Lets Decorate

Decorators with arguments

```
def tags(tag):
    def tag_decorator(func):
        def wrapper(text):
            return "<{0}>{1}</{0}>".format(tag,func(text))
        return wrapper
    return tag_decorator

@tags("p")
def get_text(text):
    return text

print(get_text("this one was complex"))
```

Lets Decorate

The name changes, it may hurt while debugging

```
'''
At the end of the day decorators are just wrapping our functions,
in case of debugging that can be problematic since the wrapper
function does not carry the name, module and docstring of the
original function.
'''
print(decoratedMessage.__name__)
```

Lets Decorate

Using functools (wraps) to annotate

```
from functools import wraps

def tagger(tag):
    def decorator(func):
        @wraps(func)
        def wrapper(text):
            return "<{0}>{1}</{0}>".format(tag,func(text))
        return wrapper
    return decorator

@tagger("div")
def get_content(text):
    return text

print(get_content("this one makes more sense"))
print(get_content.__name__)
```

Directory Ops

Python's "os" module let us interact with underlying operating system.

```
import os
# get current working directory
print(os.getcwd())

# change current working directory
os.chdir("../generators")
print(os.getcwd())

# list current directory
print(os.listdir(os.curdir))

# make directory
os.mkdir("new")
print(os.listdir(os.curdir))
```

Directory Ops...

```
# remover directory
os.rmdir("new")
print(os.listdir(os.curdir))

os.makedirs("new/1")

os.chdir("new")
print(os.listdir(os.curdir))
```

Directory Ops...

os.path lets you work with path.

```
def exists(path):
    import os.path
    return os.path.exists(path)

def isfile(path):
    import os.path
    return os.path.isfile(path=path)

print(exists("../generators1"))
print(isfile("../generators/generator.py"))
```

File Ops

Reading and Writing files

```
def move_cursor_and_read():
    text_file = open("../LICENSE", "r")
    # moves cursor by bytes/letters
    text_file.seek(20, 0)
    print(text_file.read())
    text_file.close()

def write_to_file():
    write_file = open("text.txt","w")
    write_file.write("hey dude")
    write_file.close()
```

File Ops

Reading and Writing files

```
def append_to_file():
    append_file = open("text.txt", "a+")
    append_file.write("awesome")
    append_file.close()

def read_whole_file():
    read_file = open("text.txt", "r")
    print(read_file.read())
    read_file.close()
```

File Ops

Reading and Writing files

```
def read_file():
    with open("../LICENSE", "r") as text_file:
        for line in text_file:
            print(line)

read_file()
```

File Ops

Serialization & Deserialization

```
def pickle():
    import pickle
    pickle_list = ['one', 2, 'three']
    file = open('pickled','wb')
    pickle.dump(pickle_list, file)
    file.close()

def unpickle():
    import pickle
    file = open('pickled','rb')
    unpickled_list = pickle.load(file)
    file.close()

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

Assignment

- Create a linked list or binary tree
- Add method to add element in data structure
- Print method to print data structure
- Serialize created object to a file and read it back

Recap

- Generators
- Decorators
- File and directory ops

Functional Aspects

- Buzz words about functional:
  - Immutable data
  - First class functions
  - Mapping, reducing, pipelining, recursing, currying
  - Use of higher order functions
  - Parallelization, lazy evaluation and determinism

These are all any language's features that support functional paradigm

Functional Aspects…

- Functions are first class (objects). That is, everything you can do with “data” can be done with functions themselves (such as passing a function to another function).
- Recursion is used as a primary control structure
- There is a focus on list processing
- “Pure” functional languages removes side effects.
- Functional programming either discourages or outright disallows statements, and instead works with the evaluation of expressions
- Functional programming worries about what is to be computed rather than how it is to be computed.
- Much functional programming utilizes “higher order” functions

Imperative or functional?

- Python is most definitely not a “pure functional programming language”
- Side effects are widespread in most Python programs.
- Variables are frequently rebound, mutable data collections often change contents, and I/O is freely interleaved with computation.
- It is also not even a “functional programming language” more generally
- Python is a metaparadigm language that makes functional programming easy to do when desired
- Easy to mix with other programming styles

Imperative or functional?

- Use of classes and methods to hold their imperative code
- Block of code generally consists of some outside loops (for or while)
- Assignment of state variables within those loops,
- Modification of data structures like dicts, lists, and sets
- Some branch statements (if/elif/else or try/except/finally)
- Side effects that come with state variables and mutable data structures

Think about this

- Focus not on constructing a data collection but rather on describing “what” that data collection consists of.

Encapsulation

- Focus not on constructing a data collection but rather on describing “what” that data collection consists of
- Refactor

```
# configure the data to start with
collection = get_initial_state()
state_var = None

for element in data_set:
    if condition(state_var):
        state_var = calculate_from(element)
        new = modify(element, state_var)
        collection.add_to(new)
    else:
        new = modify_differently(element)
        collection.add_to(new)

# Now actually work with the data

for thing in collection:
    process(thing)
```

```
# tuck away construction of data
def make_collection(data_set):
    collection = get_initial_state()
    state_var = None
    for datum in data_set:
        if condition(state_var):
            state_var = calculate_from(datum, state_var)
            new = modify(datum, state_var)
            collection.add_to(new)
        else:
            new = modify_differently(datum)
            collection.add_to(new)
    return collection

# Now actually work with the data

for thing in make_collection(data_set):
    process(thing)
```

Comprehensions

- A list comprehension creates a new list by applying an expression to each element of an iterable.

[<expression> for <element> in <iterable> if <condition>]

Each <element> in the <iterable> is plugged in to the <expression> if the (optional) <condition> evaluates to true

Eg: To create a list of squared integers:

```
squares = [x * x for x in (1, 2, 3, 4)]
print(squares)
```

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
```

List Comprehensions

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']
```

Dictionary Comprehensions

- Returns a dictionary

```
d = {x: x * x for x in (1, 2, 3, 4)}
d1 = dict((x, x * x) for x in (1, 2, 3, 4))
print(d, d1)
```

Generator Comprehensions

- Returns a generator using which we can iterate
- lazy

```
# list comprehension
l = [x**2 for x in range(10)]
print(l)
# generator comprehension
g = (x**2 for x in range(10))
print(g)
```

```
for i in [x**2 for x in range(10)]:
    print(i)

for i in (x**2 for x in range(10)):
    print(i)
```

What is the difference?

Set Comprehensions

- Returns a set

```
# A set containing every value in range(5):
s = {x for x in range(5)}
print(s)

# A set of even numbers between 1 and 10:
s1 = {x for x in range(1, 11) if x % 2 == 0}
print(s1)
```

Recursion

- ”What” than “How” in some way
- Limits in python recursion

```
def running_sum(numbers, start=0):
    if len(numbers) == 0:
        print()
        return
    total = numbers[0] + start
    print(total, end=" ")
    running_sum(numbers[1:], total)

print(running_sum([1,2,3,4,5]))
```

Here recursion is used for iteration

Recursion…

- Factorial in iterative and recursive

```
def factorial_recursion(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorial_recursion(N-1)


def factorial_iterative(N):
    "Iterative factorial function"
    assert isinstance(N, int) and N >= 1
    product = 1
    while N >= 1:
        product *= N
        N -= 1
    return product
```

Recursive comes closer to What than how

Recursion…

- Factorial in functional

```
def factorial_functional(n):
    return reduce(mul, range(1, n+1), 1)
```

More on this later

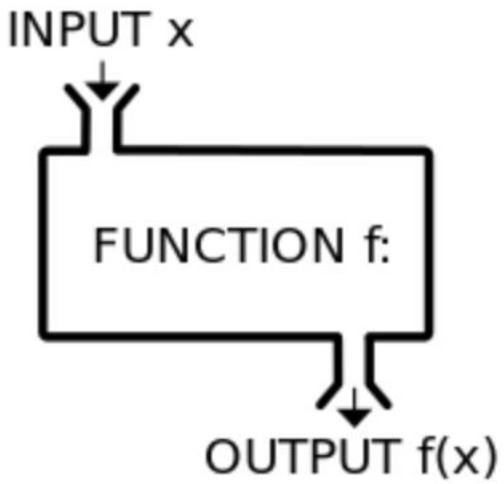
When Recursion

- Divide and conquer
- Do a similar computation on two halves
- Recursion depth is only O(log N) of the size of the collection

```
def quick_sort(lst):
    "Quick sort over a list-like sequence"
    if len(lst) == 0:
        return lst
    pivot = lst[0]
    pivots = [x for x in lst if x == pivot]
    small = quick_sort([x for x in lst if x < pivot])
    large = quick_sort([x for x in lst if x > pivot])
    print('returnning ' ,small + pivots + large)
    return small + pivots + large

l = [10,9,8,7,6,5,4,3,2,1]
print(quick_sort(l))
```

Pure function

- Functional code is characterized by one thing: the absence of side effects i.e. **pure functions**
- pure function is essentially any function that can be represented as a *mathematical expression only*
- It doesn't rely on data outside the current function
- It doesn't change data that exists outside the current function.
- Every other "functional" thing can be derived from this property



Pure function

- Output depends only on passed inputs
- Call a pure function with the same inputs a million times, you would get the same result every single time

```
# unfunctional
a = 0
def increment():
    global a
    a += 1


# functional
def increment1(a):
    return a + 1
```

Pure function…

- Where are the side effects in this code?
- x, y, xs, ys, product can be changed by some other code
- At the end these variables can be reused again with wrong values

```
# Nested loop procedural style for finding big products
xs = (1,2,3,4)
ys = (10,15,3,22)
product = []
# ...more stuff...
for x in xs:
    for y in ys:
        # ...more stuff...
        if x*y > 25:
            product.append((x,y))
            # ...more stuff...
# ...more stuff...
print(product)
```

Functions as *First-Class citizens*

- You can assign them to variables
- You can pass them as arguments
- Get them returned from other functions

```
# Functions as First-Class citizens
square_func = lambda x: x**2
# higher-order function
function_product = lambda F, m: lambda x: F(x)*m
a = function_product(square_func, 3)(2)
print(a)
```

Immutability of Data and Data Flows

- *Never modify* the value of data once initialized
- *functional* code can be thought of as a feed-forward **data flow**.
- Never ‘come back’ to change the value of any variable
- This enables usage of memoization: rememeber output of expensive function

Map

- Don’t iterate over lists. Use map.
- Takes function and collection
- Applies function to each element of collection
- Builds new collection with returned values and returns it

```
# map
names = ['Fred', 'Wilma', 'Barney']
l = list(map(len, names))
print(l)
# [4, 5, 6]
```

```
l1 = [len(item) for item in names]
print(l1)
# Out: [4, 5, 6]

l2 = (len(item) for item in names)
print(l2)
```

Map Examples

```
# take the absolute value of each element:
l3 = list(map(abs, (1, -1, 2, -2, 3, -3)))
print(l3)

# Anonymous function also support for mapping a list
l4 = map(lambda x:x*2, [1, 2, 3, 4, 5])
print(l4)


# converting decimal values to percentages
def to_percent(num):
    return num * 100
l5 = list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
print(l5)
```

Series and parallel mapping

Series mapping:

- Each argument of the iterable is supplied as argument to the mapping function
- We have just one iterable to map and the mapping function requires a single argument

Parallel mapping:

- Each argument of the mapping function is pulled from across all iterables (one from each iterable) in parallel
- Number of iterables supplied must match the number of arguments required by the function.

Series and parallel mapping…

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects)))
print(list(map(len, insects)))


carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']


def animals(x, y, z):
    return '{0}, {1} and {2} ARE ALL ANIMALS'.format(x, y, z)


print(list(map(animals, carnivores, herbivores, omnivores)))
```

Points to note

```
# Too many arguments
# observe here that map is trying to pass one item each
# from each of the four iterables to len.
# This leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
# TypeError: len() takes exactly one argument (4 given)
```

```
# Too few arguments
# observe here that map is suppose to execute animal
# on individual elements of insects one-by-one. But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
# TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Map with iterables

```
# calculating the average of each i-th element of multiple iterables
def average(*args):
    return sum(args) / len(args)

measurement1 = [100, 103, 106, 109]
measurement2 = [101, 104, 107, 110]
measurement3 = [102, 105, 108, 111]

l7 = list(map(average, measurement1, measurement2, measurement3))
print(l7)
```

```
# The mapping stops as soon as one iterable stops
measurement1 = [100, 102, 99, 97]
measurement2 = [101, 100]

# Calculate difference between elements
list(map(sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(sub, measurement2, measurement1))
# Out: [2, 6]
```

- The function must take as many parameters as there are iterables

Remarks

- Everything that can be done with map can also be done with comprehensions
- List comprehensions are efficient and can be faster than map in many cases, so test the times of both approaches if speed is important for you.

```
list(map(abs, [-1,-2,-3]))    # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]  # [1, 2, 3]

alist = [1,2,3]
list(map(add, alist, alist))  # [2, 4, 6]
l8 = [i + j for i, j in zip(alist, alist)]  # [2, 4, 6]
```

Reduce

- Performs the passed function on each subsequent item and an internal accumulator of a final result OR
- Reduce reduces an iterable by applying a function repeatedly on the next element of an iterable and the cumulative result so far
- reduce(lambda n,m:n*m, range(1,10)) means "factorial of 10”
- Multiply each item by the product of previous multiplications
- **reduce(function, iterable[, initializer])**

```
# Reduce
def add(s1, s2):
    return s1 + s2

l10 = [1, 2, 3]

sum = reduce(add, l10)
print(sum)
# equivalent to: add(add(1,2),3)
# Out: 6


# reduce can also be passed a starting value:
sum1 = reduce(add, l10, 10)
print(sum1)
```

Reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]

# Given an initializer
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30

# Without initializer
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Filter

- Uses the passed function to "evaluate" each item in a list
- Return a list of the items that pass the function test

```
# Filter
names = ['Fred', 'Wilma', 'Barney']
def long_name(name):
    return len(name) > 5

# returns a generator
filter(long_name, names)
# Out: <filter at 0x1fc6e443470>

# cast to list
l11 = list(filter(long_name, names))
print(l11)
# Out: ['Barney']

# equivalent generator expression
l12 = (name for name in names if len(name) > 5)
print(l12)
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```