Python Object-Oriented Program with Libraries

Unit 1: Object-Oriented Programming

CHAPTER 1: CLASS AND OBJECTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Inheritance

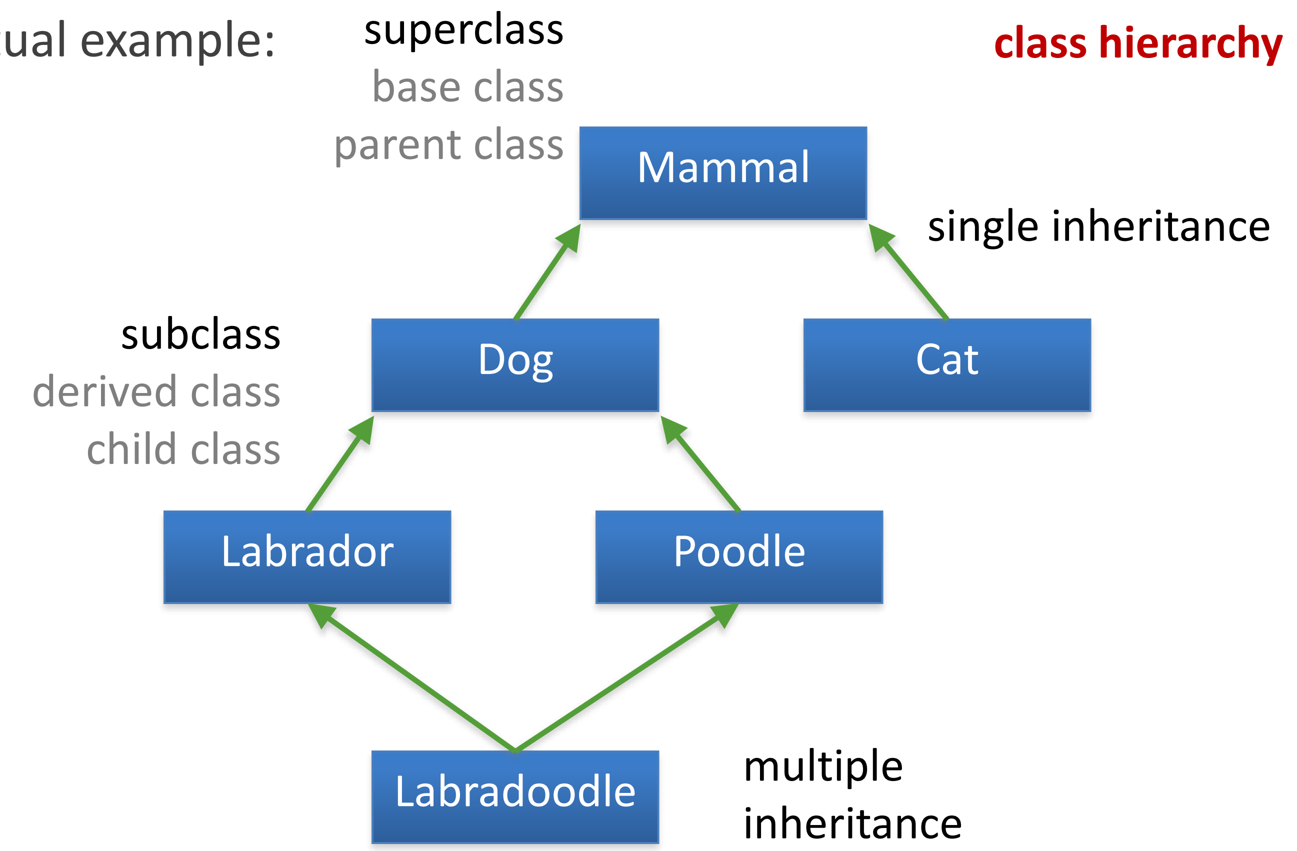
LECTURE 1

Inheritance

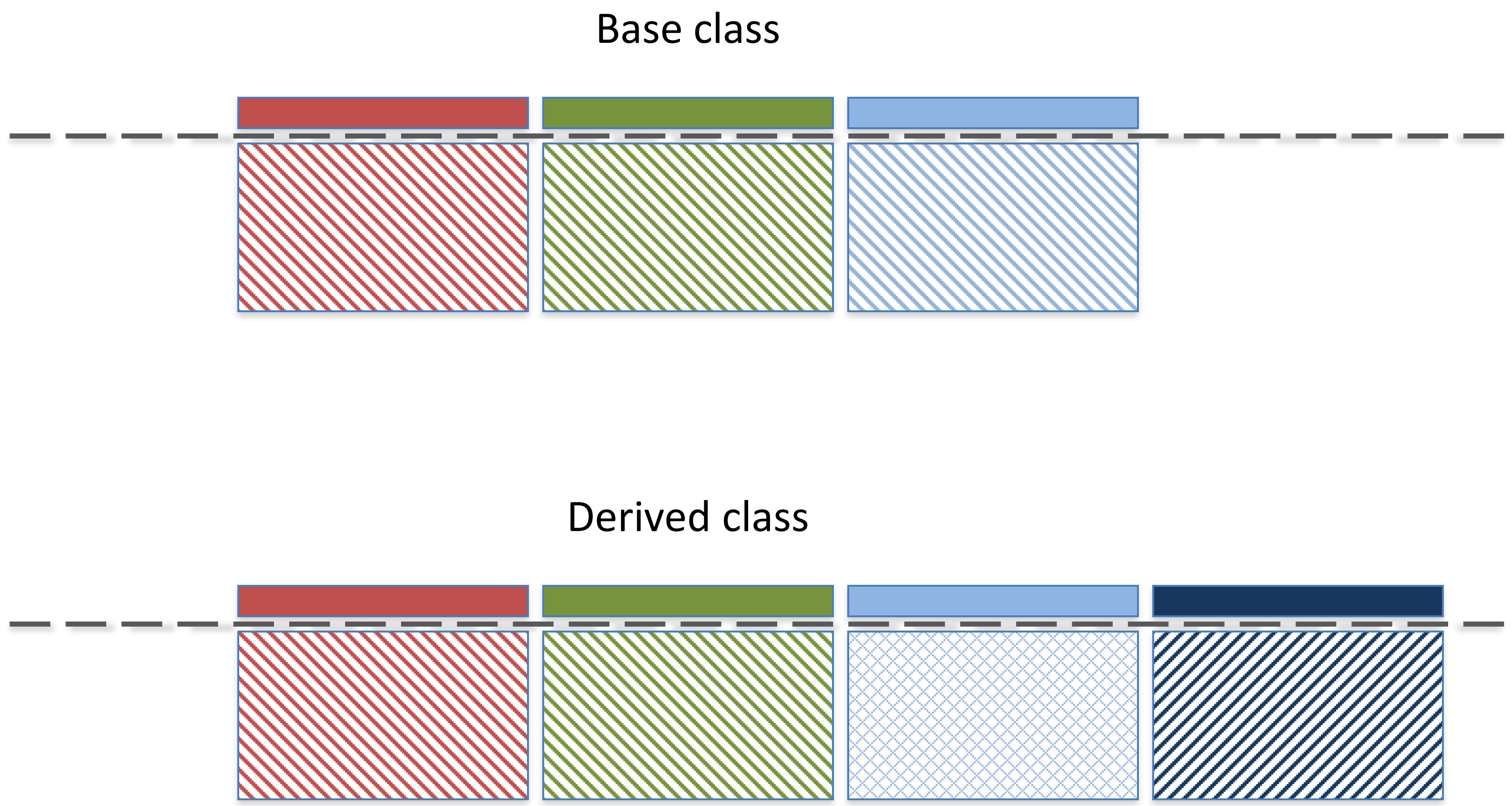
- Inheritance is the ability to define a new class that is a modified version of an existing class. – Allen Downey, *Think Python*
- "A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.
- Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more super-classes; a subclass typically augments or redefines the existing structure and behavior of superclasses." – Grady Booch, *Object-Oriented Design*

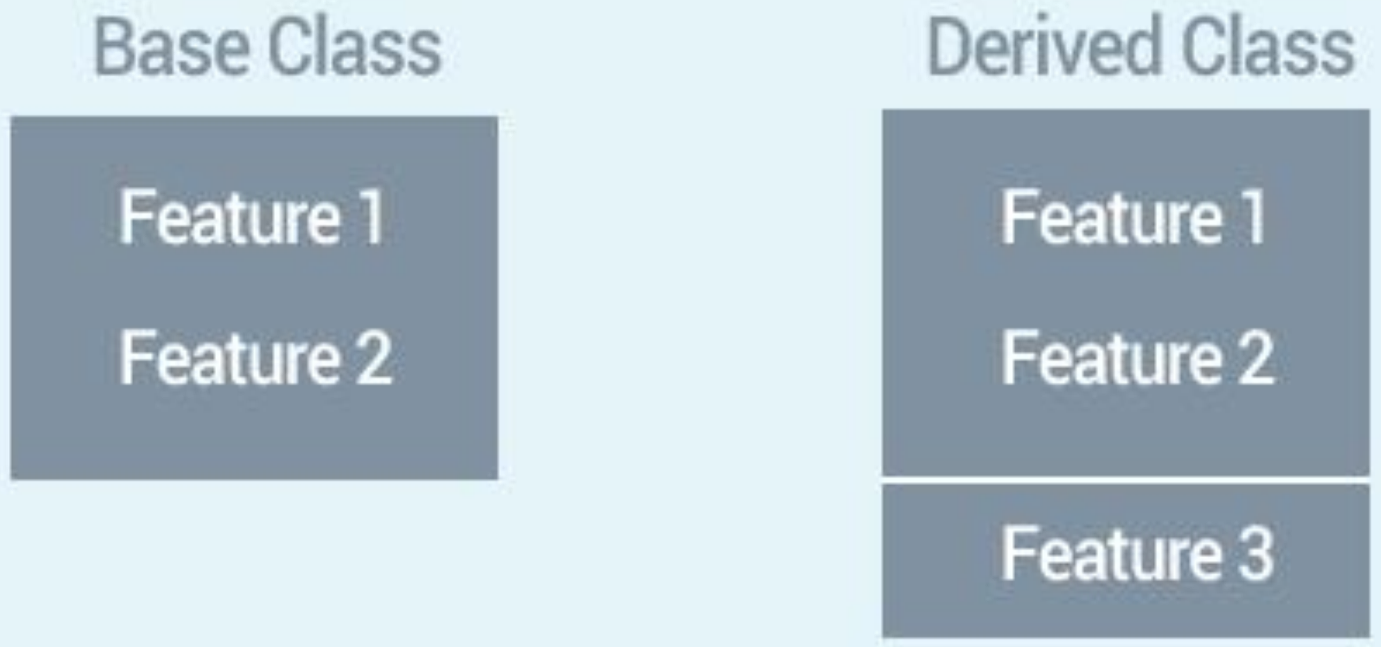
OOP Inheritance

Conceptual example:



Base Class *vs* Derived Class





Inheritance

Inheritance Syntax

The syntax for inheritance was already introduced during class declaration

- **C1** is the name of the subclass
- **object** is the name of the superclass
- for multiple inheritance, super-classes are declared as a comma-separated list of class names

```
class C1(object):
    “C1 doc”
    def f1(self):
    # do something with self
    def f2(self):
    # do something with self

# create a C1 instance
myc1 = C1()

# call f2 method
myc1.f2()
```

Inheritance

- Super-classes may be either Python- or user-defined classes
  - For example, suppose we want to use the Python list class to implement a stack (last-in, first-out) data structure
  - Python list class has a method, **pop**, for removing and returning the last element of the list
  - We need to add a **push** method to put a new element at the end of the list so that it gets popped off first

```
class Stack(list):
        “LIFO data structure"
        def push(self, element):
        self.append(element)
        # Might also have used:
        #push = list.append

st = Stack()
print("Push 12, then 1“)
st.push(12)
st.push(1)
print("Stack content", st)
print("Popping last element", st.pop())
print("Stack content now", st)
```

Inheritance Syntax

A subclass inherits all the methods of its superclass

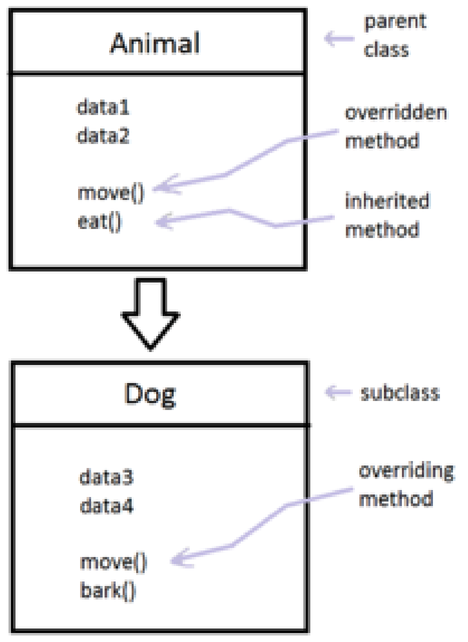
A subclass can **override** (replace or augment) methods of the superclass

- Just define a method of the same name
- Although not enforced by Python, keeping the same arguments (as well as pre- and post-conditions) for the method is highly recommended
- When augmenting a method, call the superclass method to get its functionality

A subclass can serve as the superclass for other classes

Overriding a Method

__init__ is frequently overridden because many subclasses need to both (a) let their superclass initialize their data, and (b) initialize their own data, usually in that order

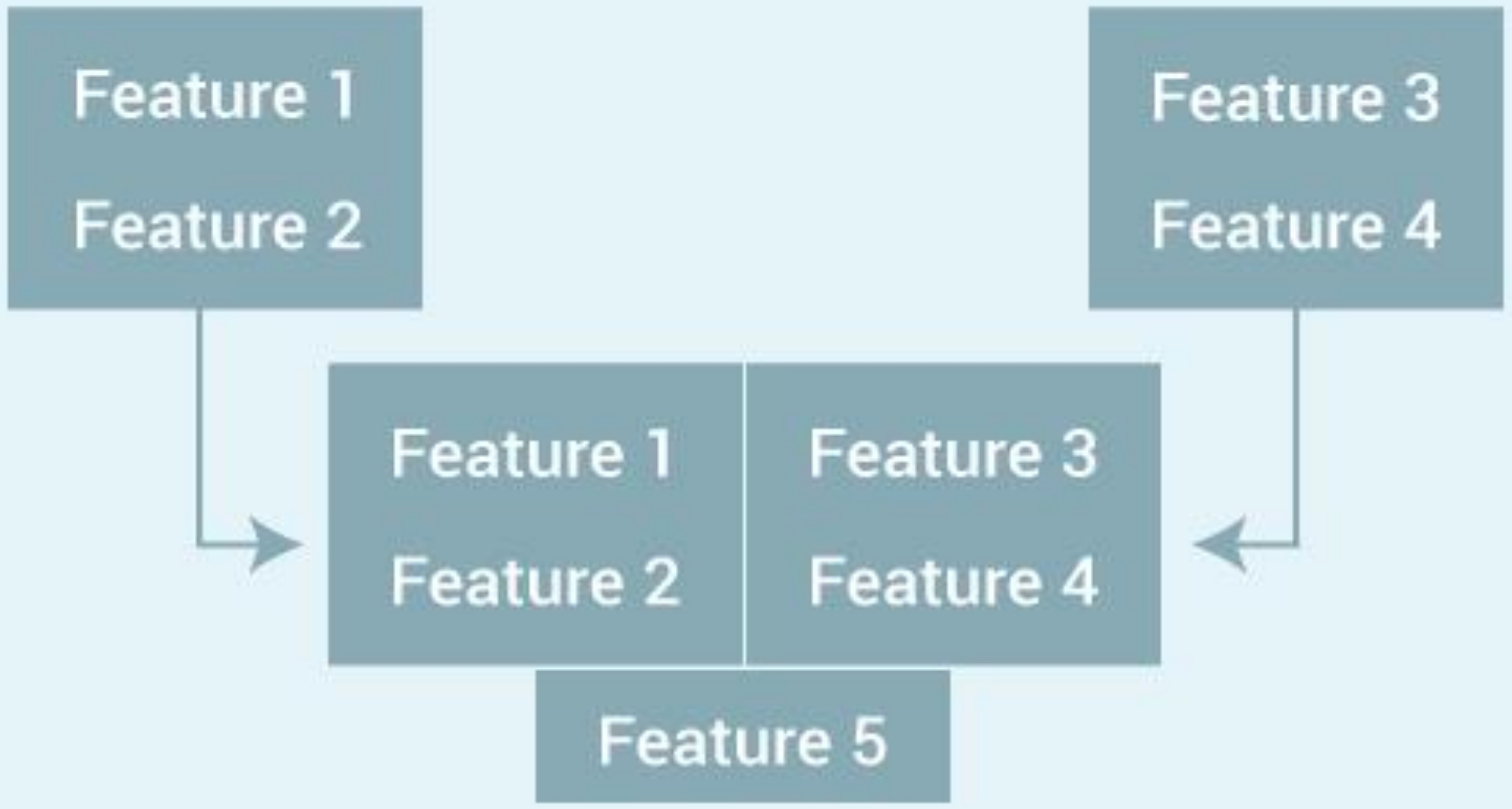


```
class Stack(list):
        push = list.append

class Calculator(Stack):
        def __init__(self):
        Stack.__init__(self)
        self.accumulator = 0
        def __str__(self):
        return str(self.accumulator)
        def push(self, value):
        Stack.push(self, value)
        self.accumulator = value

c = Calculator()
c.push(10)
print(c)
```

Mixins – Multiple Inheritance

LECTURE 1

Python Multiple Inheritance



Multiple Inheritance

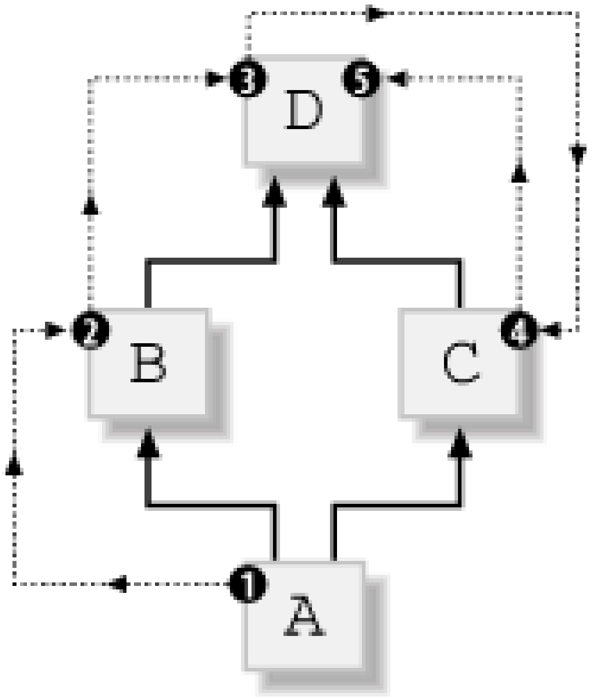
- Python supports multiple inheritance (This create chances to have collision in function names)
- In the **class** statement, replace the single superclass name with a comma-separated list of superclass names
- When looking up an attribute, Python will look for it in "Method Resolution Order" (MRO) which is approximately **left-to-right**, **depth-first** (Python 2.x older)
- There are (sometimes) subtleties that make multiple inheritance tricky to use, eg super-classes that derive from a common super-superclass
- Most of the time, single inheritance is good enough

Understanding the New Algortihm

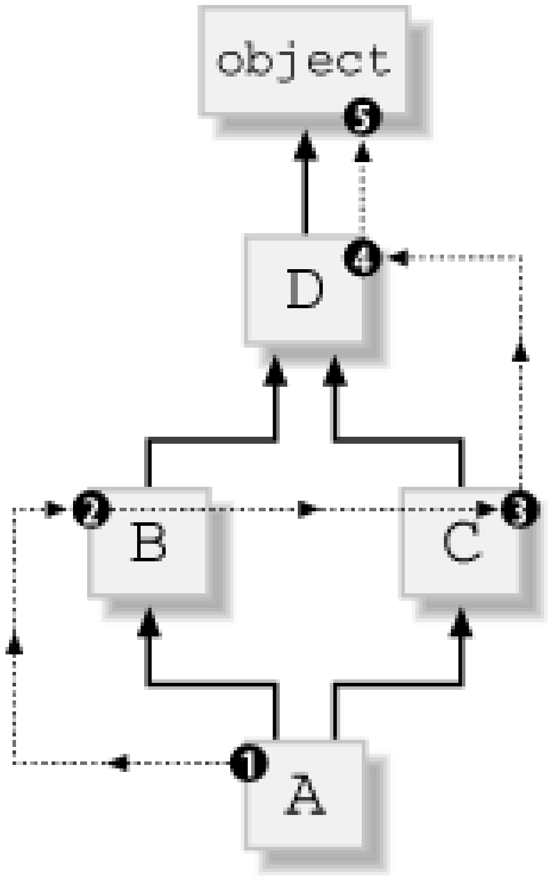
C3 Linearization

- In computing, the **C3** superclass linearization is an algorithm used primarily to obtain the order in which methods should be inherited (the "linearization") in the presence of multiple inheritance, and is often termed Method Resolution Order (**MRO**).
- The name C3 refers to the three important properties of the resulting linearization:
  - a consistent extended precedence graph,
  - preservation of local precedence order, and
  - fitting the monotonicity criterion.

Method Resolution Order



Classic method resolution order

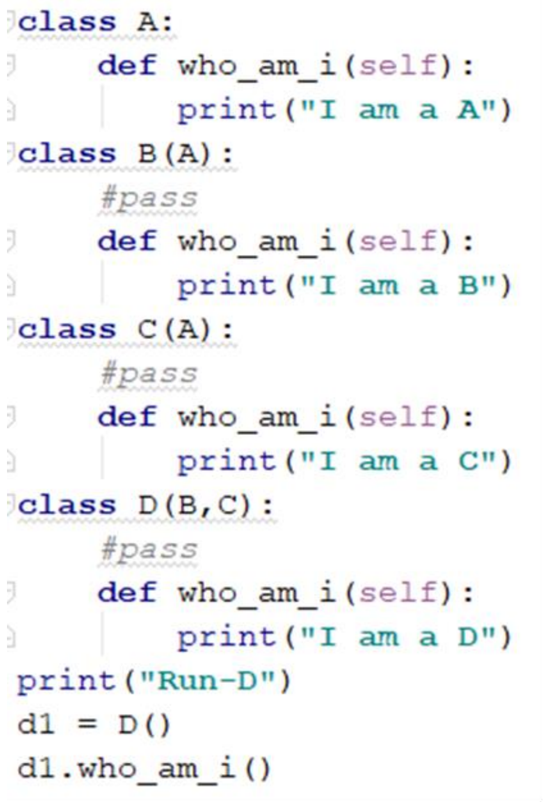


New-style method resolution order

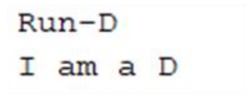
Demo Program: Python-MRO.py

Go PyCharm!!!

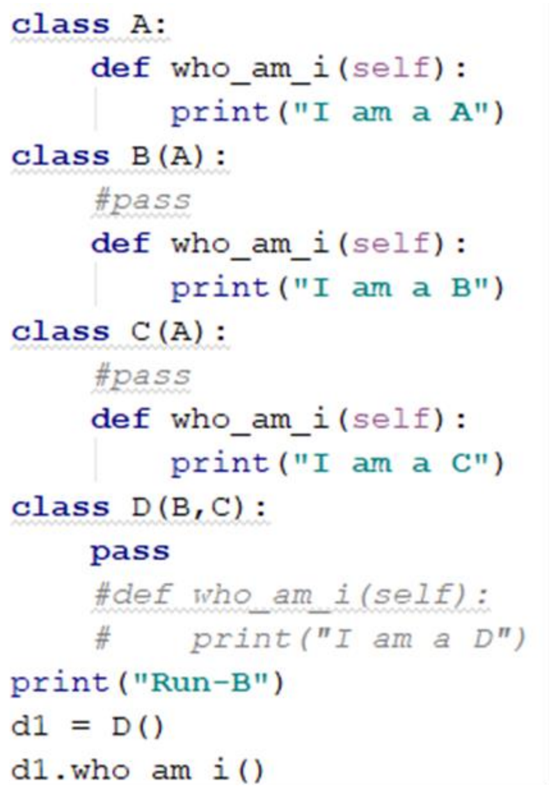
MRO Under New Algorithm



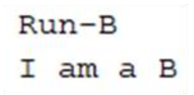
```
class A:
    def who_am_i(self):
        print("I am a A")
class B(A):
    #pass
    def who_am_i(self):
        print("I am a B")
class C(A):
    #pass
    def who_am_i(self):
        print("I am a C")
class D(B,C):
    #pass
    def who_am_i(self):
        print("I am a D")
print("Run-D")
d1 = D()
d1.who_am_i()
```



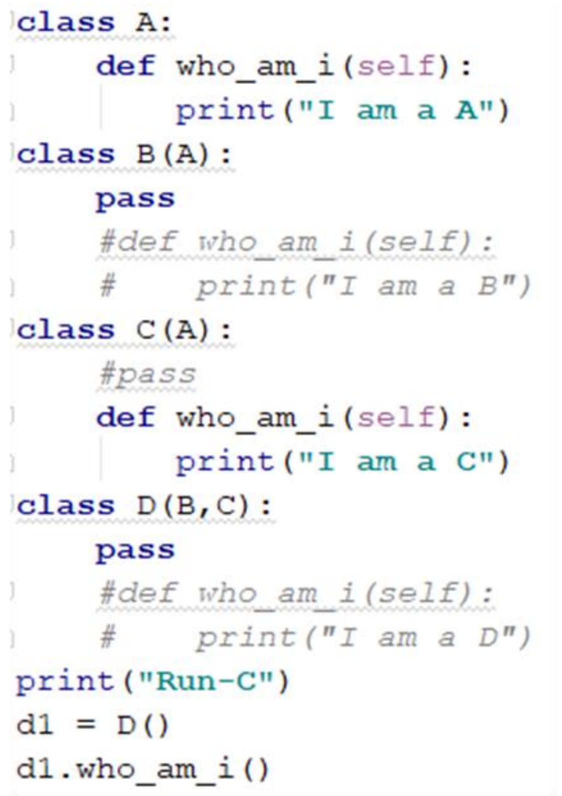
```
Run-D
I am a D
```



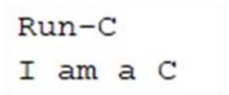
```
class A:
    def who_am_i(self):
        print("I am a A")
class B(A):
    #pass
    def who_am_i(self):
        print("I am a B")
class C(A):
    #pass
    def who_am_i(self):
        print("I am a C")
class D(B,C):
    pass
    #def who_am_i(self):
    #    print("I am a D")
print("Run-B")
d1 = D()
d1.who_am_i()
```



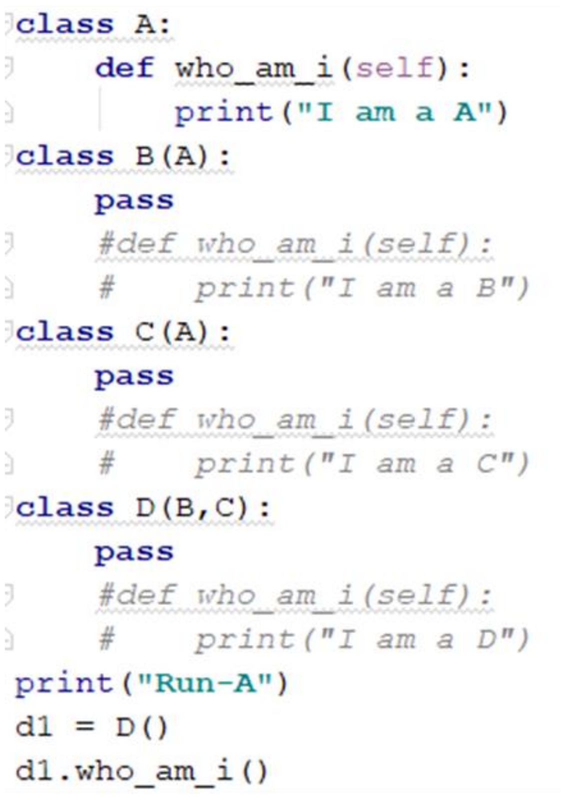
```
Run-B
I am a B
```



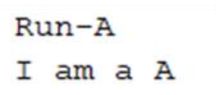
```
class A:
    def who_am_i(self):
        print("I am a A")
class B(A):
    pass
    #def who_am_i(self):
    #    print("I am a B")
class C(A):
    #pass
    def who_am_i(self):
        print("I am a C")
class D(B,C):
    pass
    #def who_am_i(self):
    #    print("I am a D")
print("Run-C")
d1 = D()
d1.who_am_i()
```



```
Run-C
I am a C
```



```
class A:
    def who_am_i(self):
        print("I am a A")
class B(A):
    pass
    #def who_am_i(self):
    #    print("I am a B")
class C(A):
    pass
    #def who_am_i(self):
    #    print("I am a C")
class D(B,C):
    pass
    #def who_am_i(self):
    #    print("I am a D")
print("Run-A")
d1 = D()
d1.who_am_i()
```

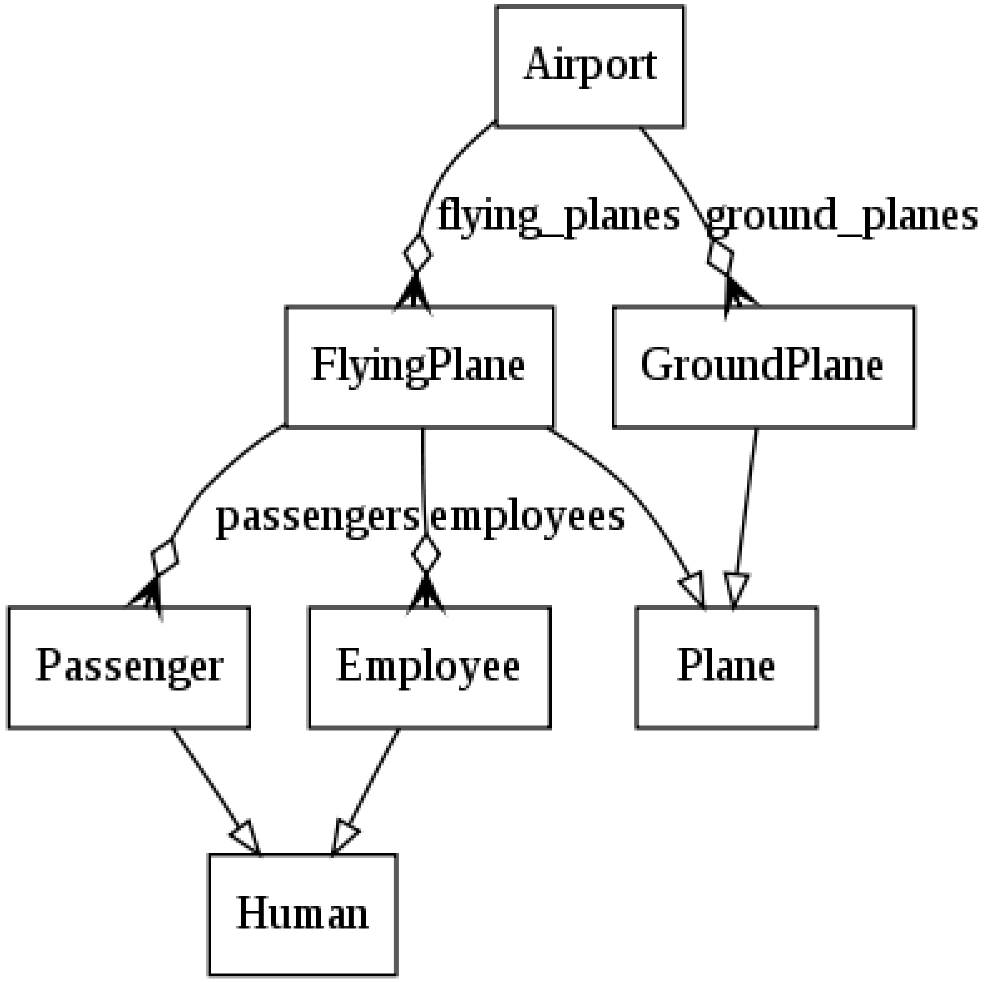


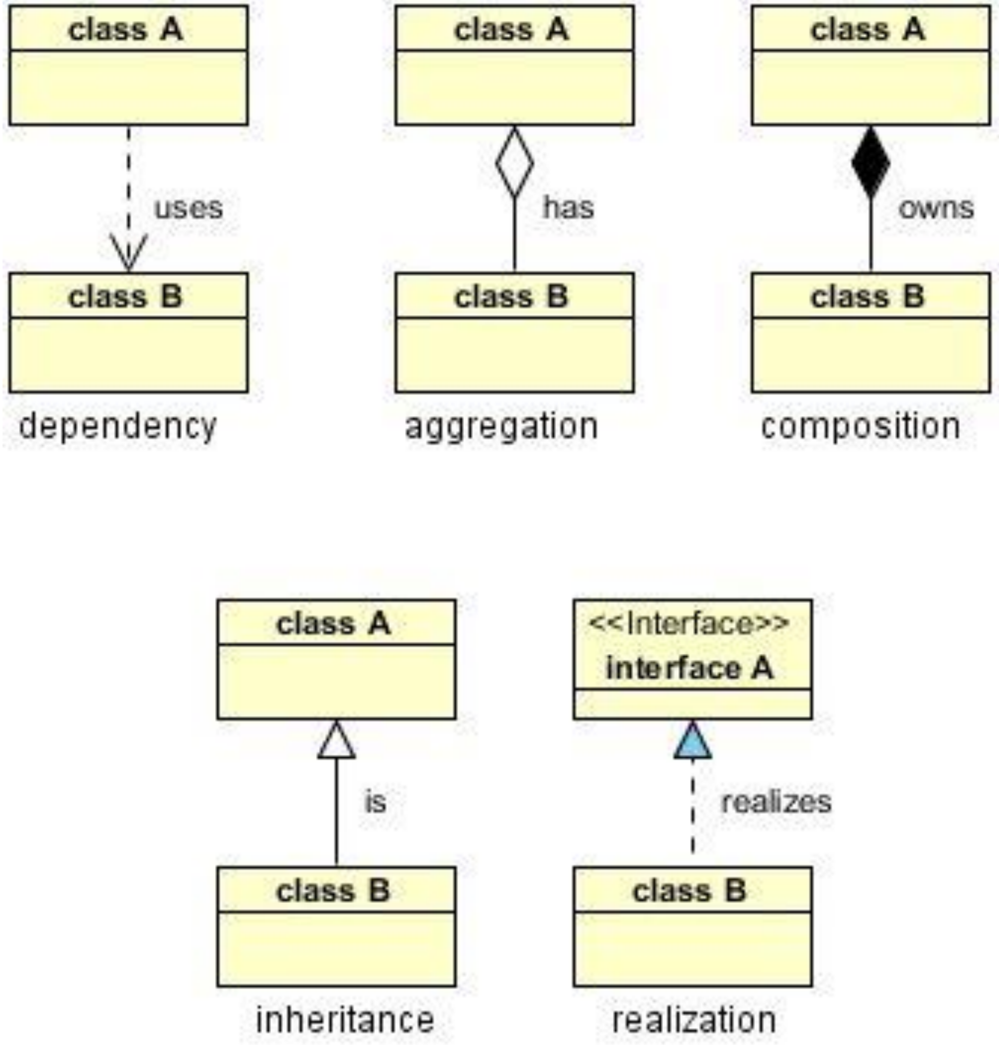
```
Run-A
I am a A
```

Class Diagrams

Class diagrams are visual representations of the relationships among classes

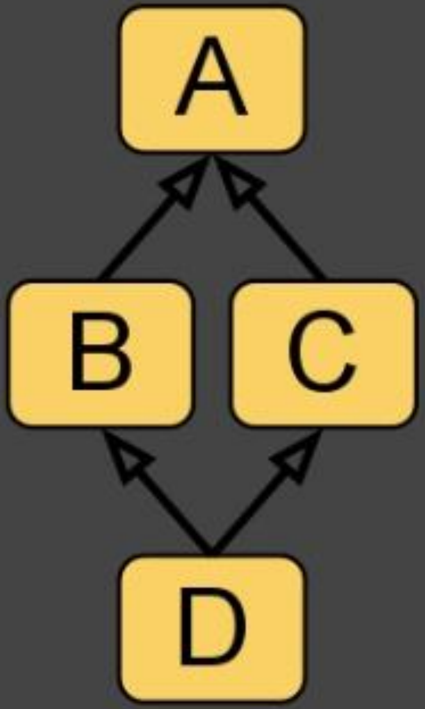
- They are similar in spirit to entity-relationship diagrams, unified modeling language, *etc* in that they help implementers in understanding and documenting application/library architecture
- They are more useful when there are more classes and attributes
- They are also very useful (along with documentation) when the code is unfamiliar


Airport
flying_planes
ground_planes
FlyingPlane
GroundPlane
passengers
employees
Passenger
Employee
Plane
Human


class A
uses
class B
dependency
class A
has
class B
aggregation
class A
owns
class B
composition
class A
is
class B
inheritance
<<Interface>>
interface A
realizes
class B
realization

Mixins (Multiple Inheritance)

```
class A(object):
    pass

class B(A):
    def method1(self):
        pass

class C(A):
    def method1(self):
        pass

class D(B, C):
    pass
```


$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Super Function

LECTURE 1

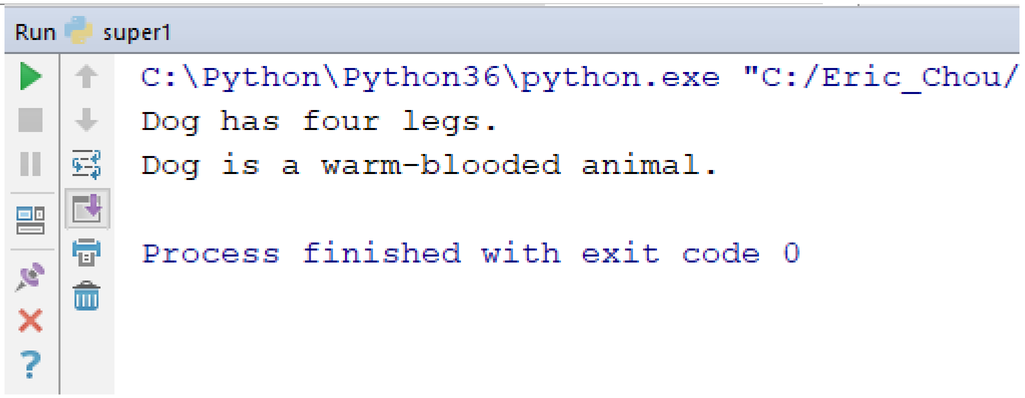
Python super()

- The **super()** builtin returns a proxy object that allows you to refer parent class by 'super'.
- In Python, **super()** built-in has two major use cases:
  1. Allows us to avoid using base class explicitly
  2. Working with Multiple Inheritance

super() with Single Inheritance Demo

Program: super1.py

Go PyCharm!!!



```
C:\Python\Python36\python.exe "C:/Eric_Chou/
Dog has four legs.
Dog is a warm-blooded animal.

Process finished with exit code 0
```

```
class Mammal(object):
    def __init__(self, mammalName):
        print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
    def __init__(self):
        print('Dog has four legs.')
        super().__init__('Dog')

d1 = Dog()
```

Use of super() to Replace the Class Name

- Here, we called __init__ method of the Mammal class (from the Dog class) using code.

  `super().__init__('Dog')`

  instead of

  `Mammal.__init__(self, 'Dog')`

- Since, we do not need to specify the name of the base class if we use super(), we can easily change the base class for Dog method easily (if we need to).

```python
# changing base class to CanidaeFamily
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # no need to change this
    super().__init__('Dog')
```

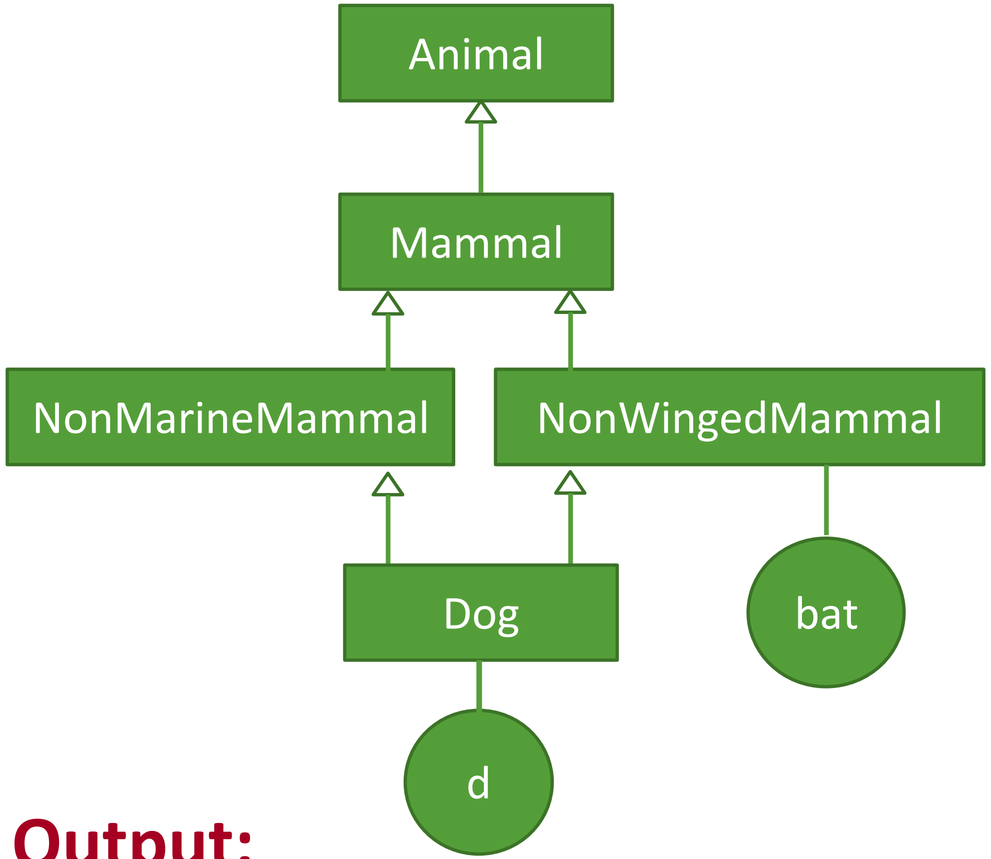
Single Inheritance

- The **super()** builtin returns a proxy object, a substitute object that has ability to call method of the base class via delegation. This is called indirection (ability to reference base object with **super()**)
- Since the indirection is computed at the **runtime**, we can use point to different base class at different time (if we need to).

super() with Multiple Inheritance
Demo Program: superm.py

Go PyCharm!!!

```python
class Animal:
    def __init__(self, animalName):
        print(animalName, 'is an animal.');
class Mammal(Animal):
    def __init__(self, mammalName):
        print(mammalName, 'is a warm-blooded animal.')
        super().__init__(mammalName)
class NonWingedMammal(Mammal):
    def __init__(self, NonWingedMammalName):
        print(NonWingedMammalName, "can't fly.")
        super().__init__(NonWingedMammalName)
class NonMarineMammal(Mammal):
    def __init__(self, NonMarineMammalName):
        print(NonMarineMammalName, "can't swim.")
        super().__init__(NonMarineMammalName)
class Dog(NonMarineMammal, NonWingedMammal):
    def __init__(self):
        print('Dog has 4 legs.');
        super().__init__('Dog')

d = Dog()
print('')
bat = NonMarineMammal('Bat')
```



Output:

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

Method Resolution Order (MRO)

- It's the order in which method should be inherited in the presence of multiple inheritance. You can view the MRO by using **__mro__** attribute.

```
(<class '__main__.Dog’>,
 <class '__main__.NonMarineMammal’>,
 <class '__main__.NonWingedMammal’>,
 <class '__main__.Mammal’>,
 <class '__main__.Animal’>,
 <class 'object'>)
```

- It shows:
  - (1) the class by MRO order;
  - (2) all the super classes.

Method Resolution Order (MRO)

Here is how **MRO** is calculated in Python:

- A method in the derived calls is always called before the method of the base class.

  In our example, Dog class is called before **NonMarineMammal** or **NoneWingedMammal**. These two classes are called before **Mammal** which is called before **Animal**, and **Animal** class is called before object.

- If there are multiple parents like **Dog(NonMarineMammal, NonWingedMammal)**, method of **NonMarineMammal** is invoked first because it appears first.

Python Command Line Input and Argument List

LECTURE 1

Python Command Line Arguments

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
C:> python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**.

This serves two purposes –

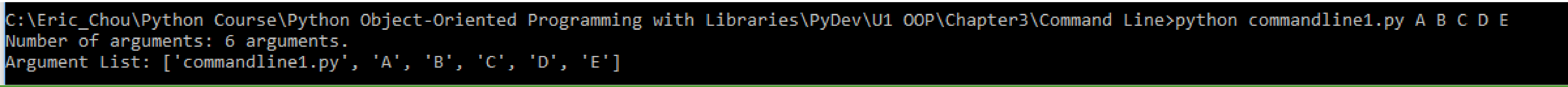
- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here **sys.argv[0]** is the program ie. script name.

Demo Program: commandline1.py

Go PyCharm!!!

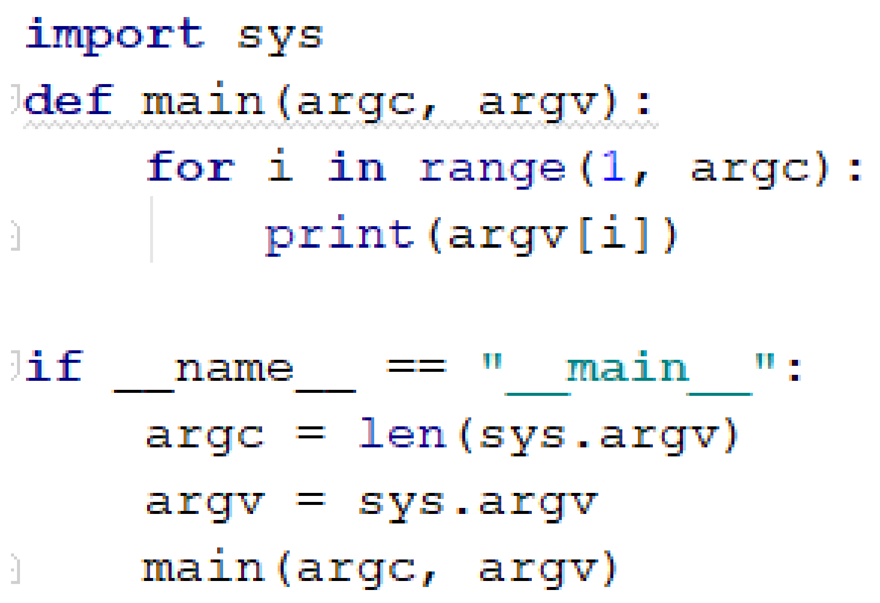
```python
import sys

print('Number of arguments:', len(sys.argv), 'arguments.')
print('Argument List:', str(sys.argv))
```



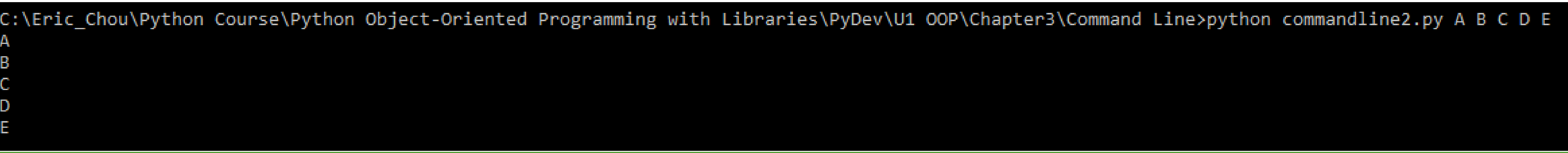
```
C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python commandline1.py A B C D E
Number of arguments: 6 arguments.
Argument List: ['commandline1.py', 'A', 'B', 'C', 'D', 'E']
```

Demo Program: commandline2.py

Go PyCharm!!!



```
import sys
def main(argc, argv):
    for i in range(1, argc):
        print(argv[i])

if __name__ == "__main__":
    argc = len(sys.argv)
    argv = sys.argv
    main(argc, argv)
```



```
C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python commandline2.py A B C D E
A
B
C
D
E
```

Learning Channel

Parsing Command-Line Arguments

- Python provided a **getopt** module that helps you parse command-line options and arguments.
- This module provides two functions and an exception to enable command line argument parsing.

getopt.getopt method
import getopt module

- This method parses command line options and parameter list. Following is simple syntax for this method –

getopt.getopt(args, options, [long_options])

getopt Processing in C Language

```c
while ((c = getopt (argc, argv, "sh:f:p:mb")) != -1)
  switch (c){
    case 's':
      suppress = 1;
      break;
    case 'h':
      opth = atoi(optarg);
      if (opth>0) ht_SIZE = opth;
      break;
    case 'f':
      optf = atoi(optarg);
      if (optf>0) bloomF_SIZE = optf;
      break;
    case 'p':
      optp = atoi(optarg);
      if (optp>0) pCount = optp;
      break;
    case 'm':
      moveToFront =1;
      break;
    case 'b':
      moveToFront=0;
      break;
    default:
      printf("Some program argument setting error!\n");
      printf("Usage: banhammer \n");
      printf("                 -s             : will suppress the letter from the censor, and instead print the statistics that were computed.\n");
      printf("                 -h size     : size specifies that the bash table will have size entries.    [default=1000]\n");
      printf("                 -f  size     : size specifies that the Bloom filter will have size entries. [default=2^20]\n");
      printf("                 -m            : will suppress the letter from the censor, and instead print the statistics that were computed.\n");
      printf("                 -b             : will suppress the letter from the censor, and instead print the statistics that were computed.\n");
      printf("                 -p  num  : number of bit vectors to be printed.\n");
      break;
    }
```

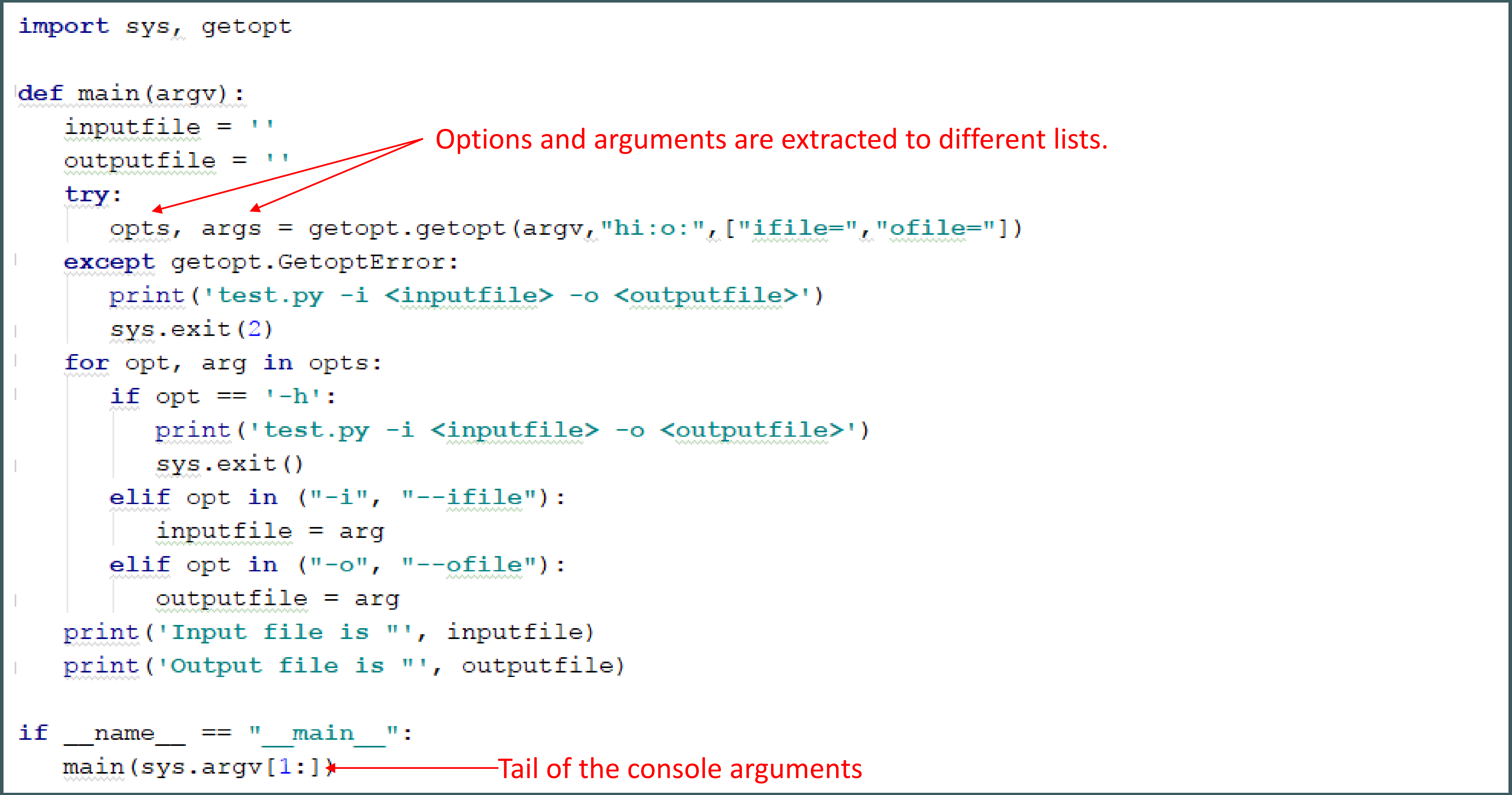
Each time an option c is fetched, optarg is updated.
(Non-option characters are returned in argv.)

Learning Channel

getopt.getopt method

Here is the detail of the parameters –

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.
- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').



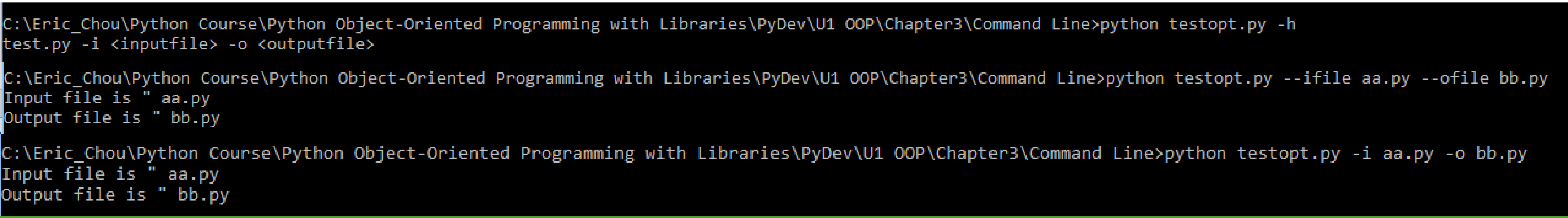
```
import sys, getopt

def main(argv):
    inputfile = ''
    outputfile = ''
    try:
        opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
    except getopt.GetoptError:
        print('test.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print('test.py -i <inputfile> -o <outputfile>')
            sys.exit()
        elif opt in ("-i", "--ifile"):
            inputfile = arg
        elif opt in ("-o", "--ofile"):
            outputfile = arg
    print('Input file is "', inputfile)
    print('Output file is "', outputfile)

if __name__ == "__main__":
    main(sys.argv[1:])
```

Options and arguments are extracted to different lists.

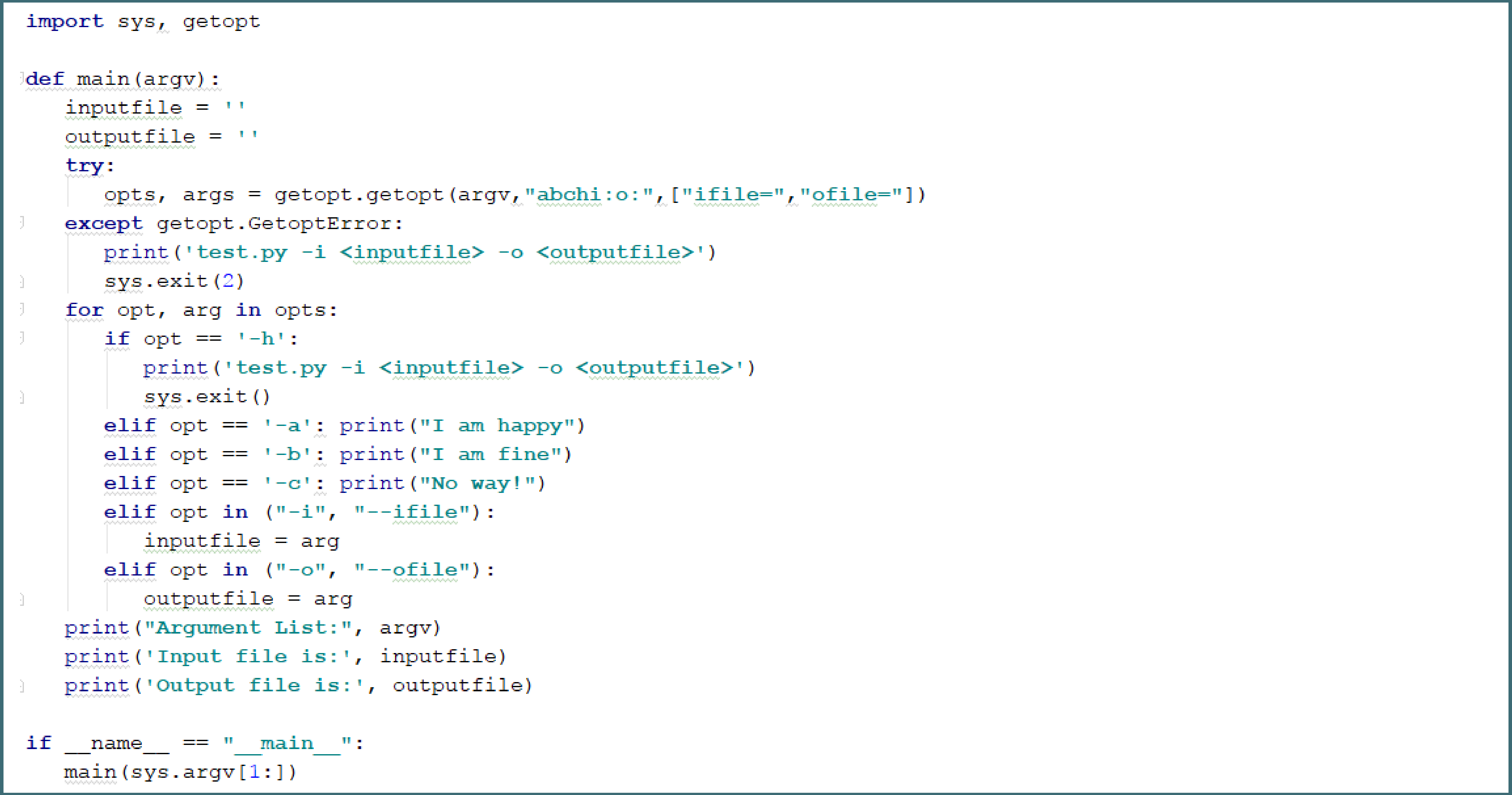
Tail of the console arguments

Demo Program: testopt.py

Go PyCharm!!!



```
C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python testopt.py -h
test.py -i <inputfile> -o <outputfile>

C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python testopt.py --ifile aa.py --ofile bb.py
Input file is " aa.py
Output file is " bb.py

C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python testopt.py -i aa.py -o bb.py
Input file is " aa.py
Output file is " bb.py
```

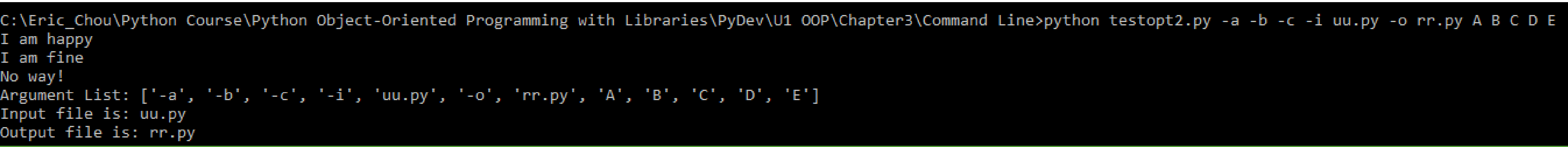


```
import sys, getopt

def main(argv):
   inputfile = ''
   outputfile = ''
   try:
      opts, args = getopt.getopt(argv,"abchi:o:",["ifile=","ofile="])
   except getopt.GetoptError:
      print('test.py -i <inputfile> -o <outputfile>')
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print('test.py -i <inputfile> -o <outputfile>')
         sys.exit()
      elif opt == '-a': print("I am happy")
      elif opt == '-b': print("I am fine")
      elif opt == '-c': print("No way!")
      elif opt in ("-i", "--ifile"):
         inputfile = arg
      elif opt in ("-o", "--ofile"):
         outputfile = arg
   print("Argument List:", argv)
   print('Input file is:', inputfile)
   print('Output file is:', outputfile)

if __name__ == "__main__":
   main(sys.argv[1:])
```

Demo Program: testopt2.py

Go PyCharm!!!

- -i short form for option
- -ifile long form for option
- -a no argument option
- Non-optional arguments after optional arguments. (‘A’, ‘B’, ‘C’, ‘D’, ‘E’)
- No non-optional arguments before the optional arguments.



```
C:\Eric_Chou\Python Course\Python Object-Oriented Programming with Libraries\PyDev\U1 OOP\Chapter3\Command Line>python testopt2.py -a -b -c -i uu.py -o rr.py A B C D E
I am happy
I am fine
No way!
Argument List: ['-a', '-b', '-c', '-i', 'uu.py', '-o', 'rr.py', 'A', 'B', 'C', 'D', 'E']
Input file is: uu.py
Output file is: rr.py
```

Magic Variables

LECTURE 1

*args and **kwargs

- I have come to see that most new python programmers have a hard time figuring out the ***args** and ****kwargs** magic variables. So what are they ?
- First of all let me tell you that it is not necessary to write ***args** or ****kwargs**.
- Only the * (asterisk) is necessary. You could have also written ***var** and ****vars**.
- Writing ***args** and ****kwargs** is just a convention. So now lets take a look at ***args** first.

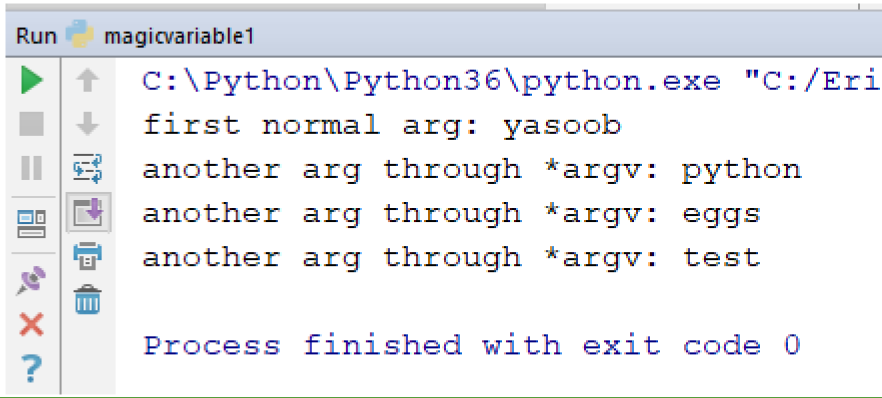
Usage of *args

Variable number of arguments

- *args and **kwargs are mostly used in function definitions. *args and **kwargs allow you to pass a **variable** number of arguments to a function. What variable means here is that you do not know beforehand how many arguments can be passed to your function by the user so in this case you use these two keywords.
- *args is used to send a non-keyworded variable length argument list to the function. Here’s an example to help you get a clear idea:

Demo Program: magicvariable1.py

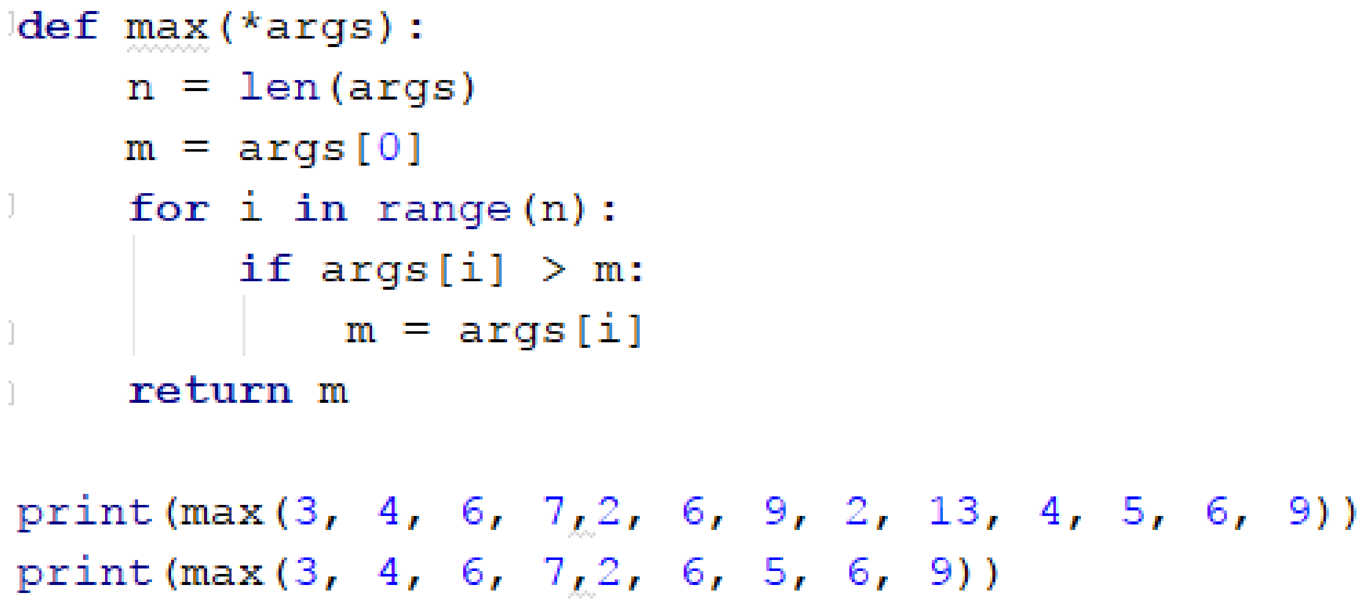
```python
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```



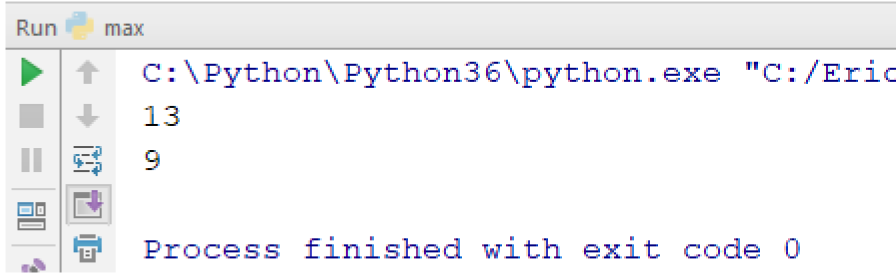
```
C:\Python\Python36\python.exe "C:/Eri
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test

Process finished with exit code 0
```

Demo Program: max.py

- Finding the maximum number in a group of data with variable group size.



```python
def max(*args):
    n = len(args)
    m = args[0]
    for i in range(n):
        if args[i] > m:
            m = args[i]
    return m

print(max(3, 4, 6, 7,2, 6, 9, 2, 13, 4, 5, 6, 9))
print(max(3, 4, 6, 7,2, 6, 5, 6, 9))
```

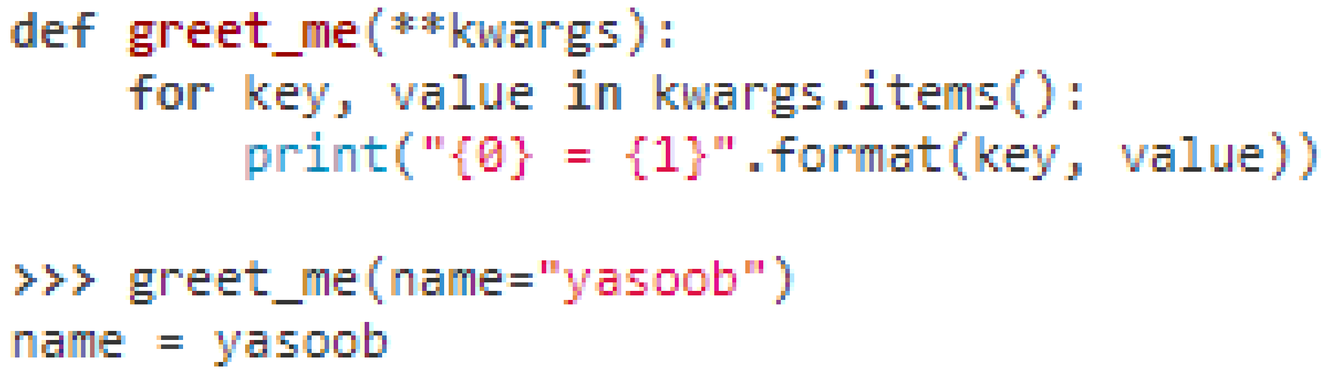


```
C:\Python\Python36\python.exe "C:/Eric
13
9

Process finished with exit code 0
```

Usage of **kwargs

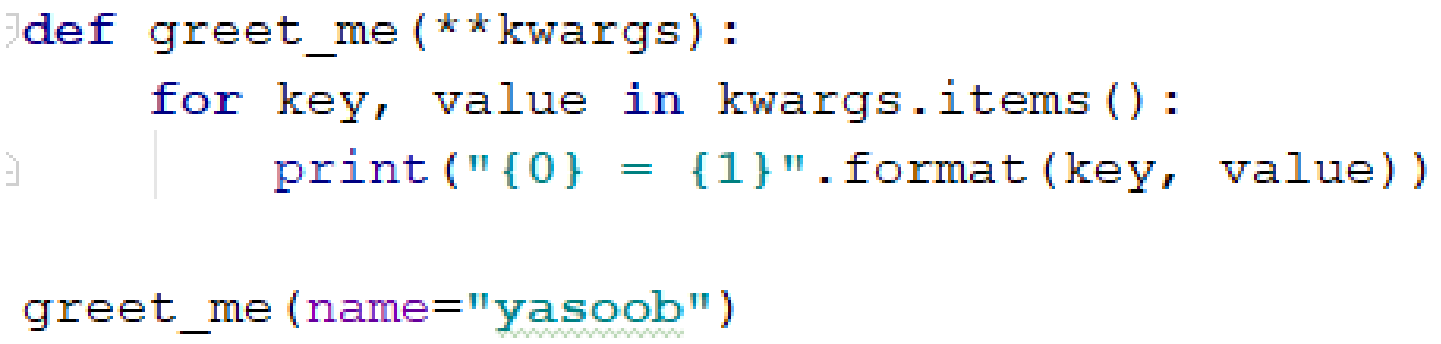
variable number of keyworded arguments

- **kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function. Here is an example to get you going with it:

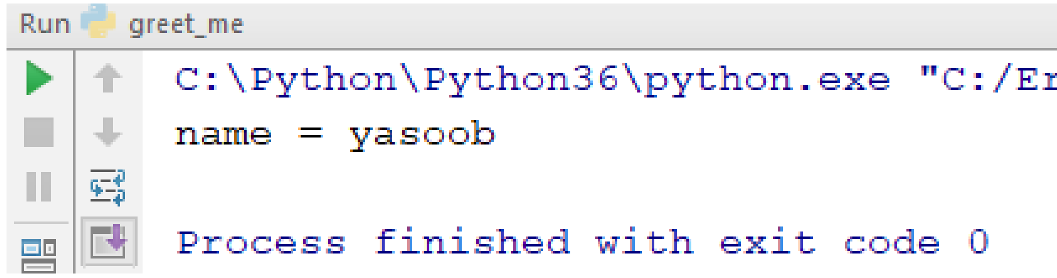


```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

>>> greet_me(name="yasoob")
name = yasoob
```

Demo Program: greet_me.py



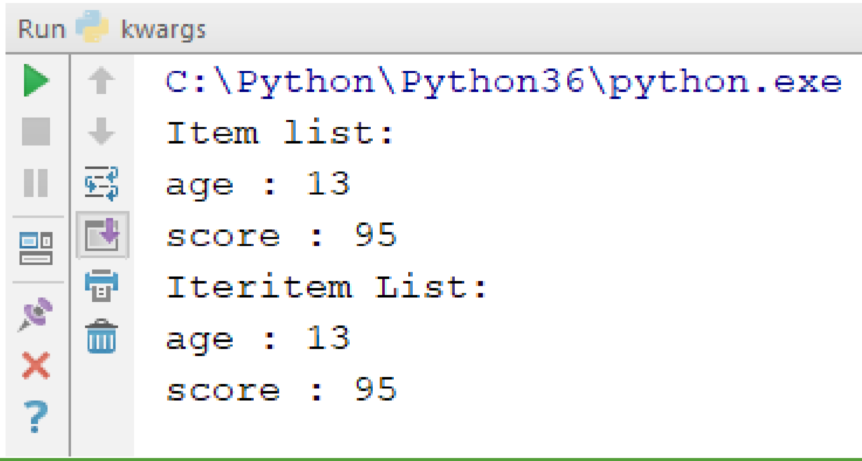
```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

greet_me(name="yasoob")
```



```
C:\Python\Python36\python.exe "C:/Er
name = yasoob

Process finished with exit code 0
```

Demo Program: kwargs.py

```
def func(**kwargs):
    print("Item list:")
    for (key, value) in kwargs.items():
        print("%s : %d" % (key, value))
    print("Iteritem List:")
    for t in kwargs.items():
        print("%s : %d" % (t[0], t[1]))

func(age=13, score=95)
```



```
C:\Python\Python36\python.exe
Item list:
age : 13
score : 95
Iteritem List:
age : 13
score : 95
```

Using *args and **kwargs to call a function

Call-by-tuple **f(*args)**

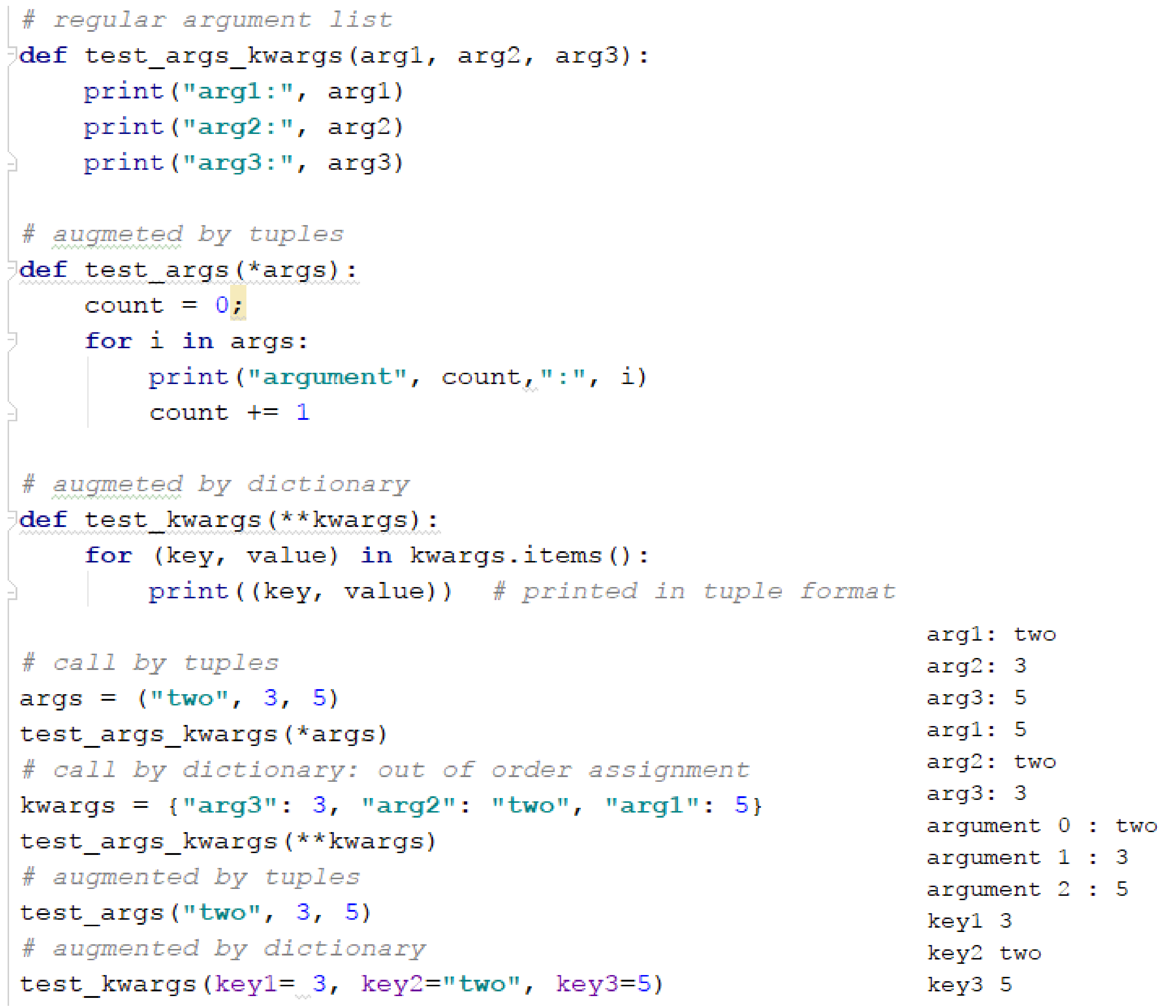
Call-by-dictionry **f(**kwargs)**

Augmented by tuple

  def f(*args)

Augmented by dictionary

  def f(kwargs)**



```python
# regular argument list
def test_args_kwargs(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)

# augmeted by tuples
def test_args(*args):
    count = 0;
    for i in args:
        print("argument", count,":", i)
        count += 1

# augmeted by dictionary
def test_kwargs(**kwargs):
    for (key, value) in kwargs.items():
        print((key, value))  # printed in tuple format

# call by tuples
args = ("two", 3, 5)
test_args_kwargs(*args)
# call by dictionary: out of order assignment
kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
test_args_kwargs(**kwargs)
# augmented by tuples
test_args("two", 3, 5)
# augmented by dictionary
test_kwargs(key1= 3, key2="two", key3=5)
```

```
arg1: two
arg2: 3
arg3: 5
arg1: 5
arg2: two
arg3: 3
argument 0 : two
argument 1 : 3
argument 2 : 5
key1 3
key2 two
key3 5
```

When to use them?

- It really depends on what your requirements are. The most common use case is when making function decorators.
- Moreover it can be used in monkey patching as well. Monkey patching means modifying some code at runtime.
- Consider that you have a class with a function called **get_info** which calls an API and returns the response data. If we want to test it we can replace the API call with some test data. For instance:

```python
import someclass

def get_info(self, *args):
    return "Test data"

someclass.get_info = get_info
```

Super Function in Details

LECTURE 1

How Is the Super Function Used?

- The super function is somewhat versatile, and can be used in a couple of ways.
  - **Use Case 1:** Super can be called upon in a single inheritance, in order to refer to the parent class or multiple classes without explicitly naming them. It's somewhat of a shortcut, but more importantly, it helps keep your code maintainable for the foreseeable future.
  - **Use Case 2:** Super can be called upon in a dynamic execution environment for multiple or collaborative inheritance. This use is considered exclusive to Python, because it's not possible with languages that only support single inheritance or are statically compiled.

How Is the Super Function Used?

- The great thing about super is that it can be used to enhance any module method. Plus, there’s no need to know the details about the base class that’s being used as an extender. The super function handles all of it for you.
- So, for all intents and purposes, **super** is a shortcut to access a base class without having to know its **type** or **name**.

Syntax for Super()

In Python 3 and above, the syntax for super is:

```
super().methoName(args)
```

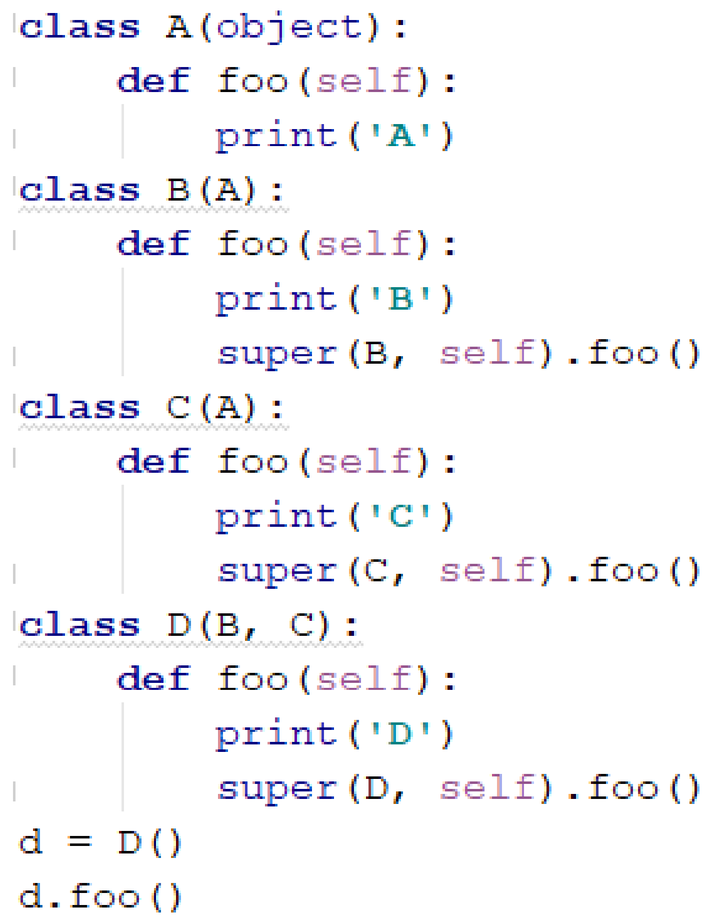
Whereas the normal way to call super (in older builds of Python) is:

```
super(subClass, instance).method(args)
```

Use the MRO algorithm to find the super class of this subclass and the specific object .

Demo Program: python_mro2.py

Go PyCharm!!!


```
class A(object):
    def foo(self):
        print('A')
class B(A):
    def foo(self):
        print('B')
        super(B, self).foo()
class C(A):
    def foo(self):
        print('C')
        super(C, self).foo()
class D(B, C):
    def foo(self):
        print('D')
        super(D, self).foo()
d = D()
d.foo()
```

Dynamic Binding of Function

- super() is in the business of delegating method calls to some class in the instance’s ancestor tree. For reorderable method calls to work, the classes need to be designed cooperatively. This presents three easily solved practical issues:
  - the method being called by **super()** needs to exist
  - the **caller** and **callee** need to have a matching argument signature
  - and every occurrence of the method needs to use **super()**

Binding with Proper Positional Arguments

- One approach is to stick with a fixed signature using positional arguments. This works well with methods like which have a fixed signature of two arguments, a key and a value.

Binding with Flexible Argument List

A more flexible approach is to have every method in the ancestor tree cooperatively designed to accept keyword arguments and a keyword-arguments dictionary, to remove any arguments that it needs, and to forward the remaining arguments using ****kwds**, eventually leaving the dictionary empty for the final call in the chain.

Binding with Flexible Argument List

Each level strips-off the keyword arguments that it needs so that the final empty dict can be sent to a method that expects no arguments at all (for example, **object.__init__** expects zero arguments):

```
class Shape:
    def __init__(self, shapename, **kwds):
        self.shapename = shapename
        super().__init__(**kwds)

class ColoredShape(Shape):
    def __init__(self, color, **kwds):
        self.color = color
        super().__init__(**kwds)


cs = ColoredShape(color='red', shapename='circle')
```

Abstract Method

LECTURE 1

Module abc — Abstract Base Classes

Purpose: Define and use abstract base classes for interface verification.

Why use Abstract Base Classes?¶

- Abstract base classes are a form of interface checking more strict than individual **hasattr()** checks for particular methods.
- By defining an abstract base class, a common API can be established for a set of subclasses.
- This capability is especially useful in situations where someone less familiar with the source for an application is going to provide plug-in extensions, but can also help when working on a large team or with a large code-base where keeping track of all of the classes at the same time is difficult or not possible.

Abstract Base Classes

- This module provides the infrastructure for defining abstract base classes (ABCs) in Python.
- The collections module has some concrete classes that derive from ABCs; these can, of course, be further derived.
- In addition the **collections.abc** submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, is it **hashable** or a mapping.
- **Interface:** Abstract Classes with only abstract methods.
- **Abstract Class**: Classes has data fields, concrete methods and abstract methods.
- Python's Multiple-Inheritance can accommodate all Interface and Abstract Classes. There is no program structure for Interface and abstract classes. They are just classes.

How ABCs Work

abc works by marking methods of the base class as abstract, and then registering concrete classes as implementations of the abstract base. If your code requires a particular API, you can use **issubclass()** or **isinstance()** to check an object against the abstract class.

Let’s start by defining an abstract base class to represent the API of a set of plugins for saving and loading data.

class abc.ABC

Two Ways to Create Abstract Base Class

- A helper class that has **ABCMeta** as its **metaclass**. With this class, an abstract base class can be created by simply deriving from **ABC** avoiding sometimes confusing **metaclass** usage, for example:

```
from abc import ABC
class MyABC(ABC):
    pass
```

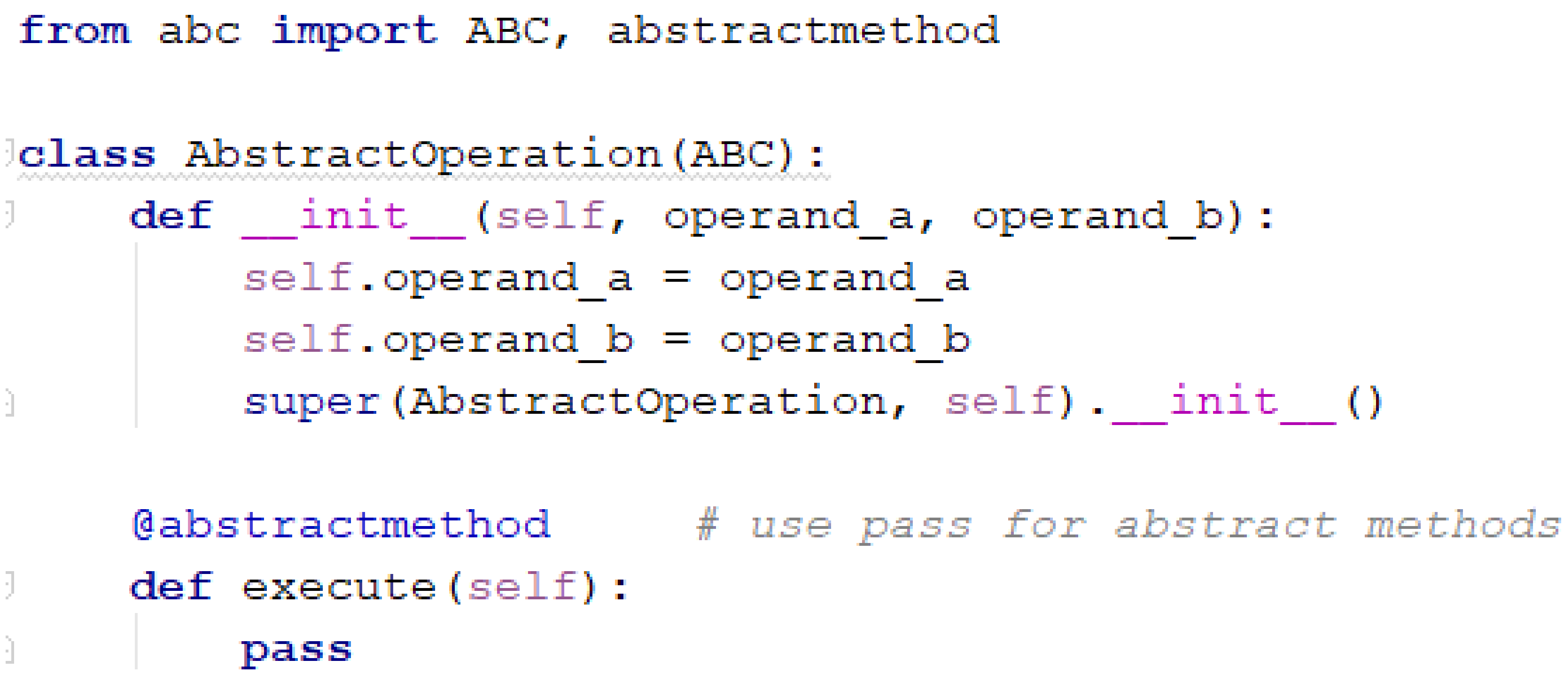
- Note that the type of ABC is still **ABCMeta**, therefore inheriting from ABC requires the usual precautions regarding **metaclass** usage, as multiple inheritance may lead to **metaclass** conflicts.

- One may also define an abstract base class by passing the **metaclass** keyword and using **ABCMeta** directly, for example:

```
from abc import ABCMeta
class MyABC(metaclass=ABCMeta):
        pass
```

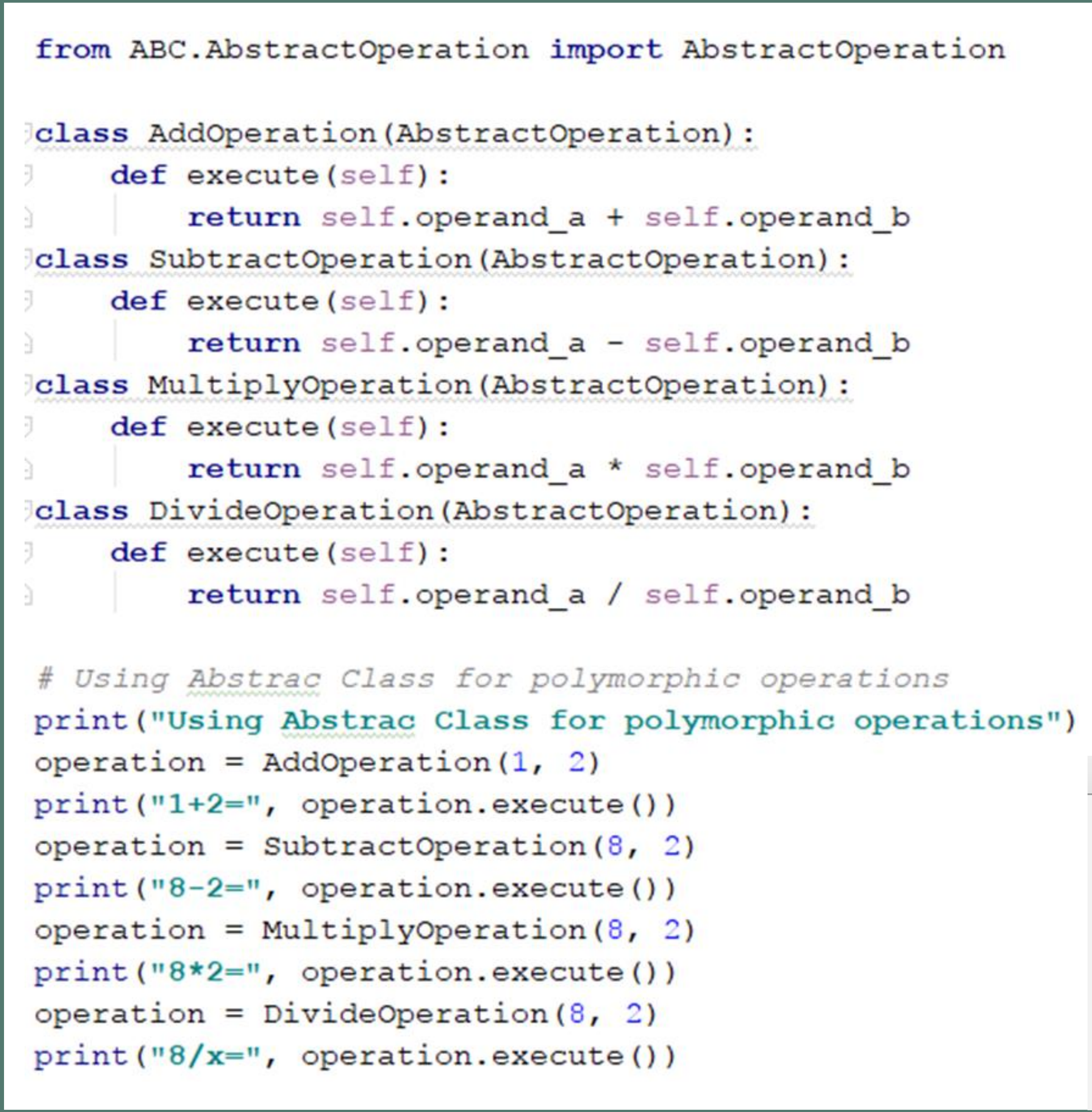
How ABCs Work?

Create Abstract Method (AbstractOperation.py)

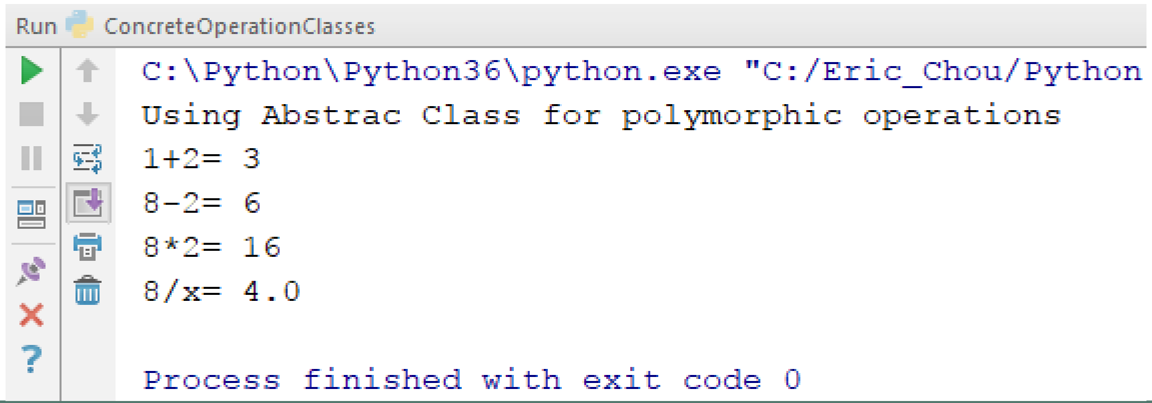


```python
from abc import ABC, abstractmethod

class AbstractOperation(ABC):
    def __init__(self, operand_a, operand_b):
        self.operand_a = operand_a
        self.operand_b = operand_b
        super(AbstractOperation, self).__init__()

    @abstractmethod     # use pass for abstract methods
    def execute(self):
        pass
```

Demo Program: ConcreteOperations.py



```python
from ABC.AbstractOperation import AbstractOperation

class AddOperation(AbstractOperation):
    def execute(self):
        return self.operand_a + self.operand_b
class SubtractOperation(AbstractOperation):
    def execute(self):
        return self.operand_a - self.operand_b
class MultiplyOperation(AbstractOperation):
    def execute(self):
        return self.operand_a * self.operand_b
class DivideOperation(AbstractOperation):
    def execute(self):
        return self.operand_a / self.operand_b

# Using Abstrac Class for polymorphic operations
print("Using Abstrac Class for polymorphic operations")
operation = AddOperation(1, 2)
print("1+2=", operation.execute())
operation = SubtractOperation(8, 2)
print("8-2=", operation.execute())
operation = MultiplyOperation(8, 2)
print("8*2=", operation.execute())
operation = DivideOperation(8, 2)
print("8/x=", operation.execute())
```



```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python
Using Abstrac Class for polymorphic operations
1+2= 3
8-2= 6
8*2= 16
8/x= 4.0

Process finished with exit code 0
```

Declaring Abstract Base Class

1. Inherit **ABC** class and **abstractmethod** method

2. At concrete class, inherit the customer-defined Abstract Base Class

3. An Abstract Class without data field is an **Interface**.

Abstract Geometric Object Class Demo Program: geometry.py

Go PyCharm!!!

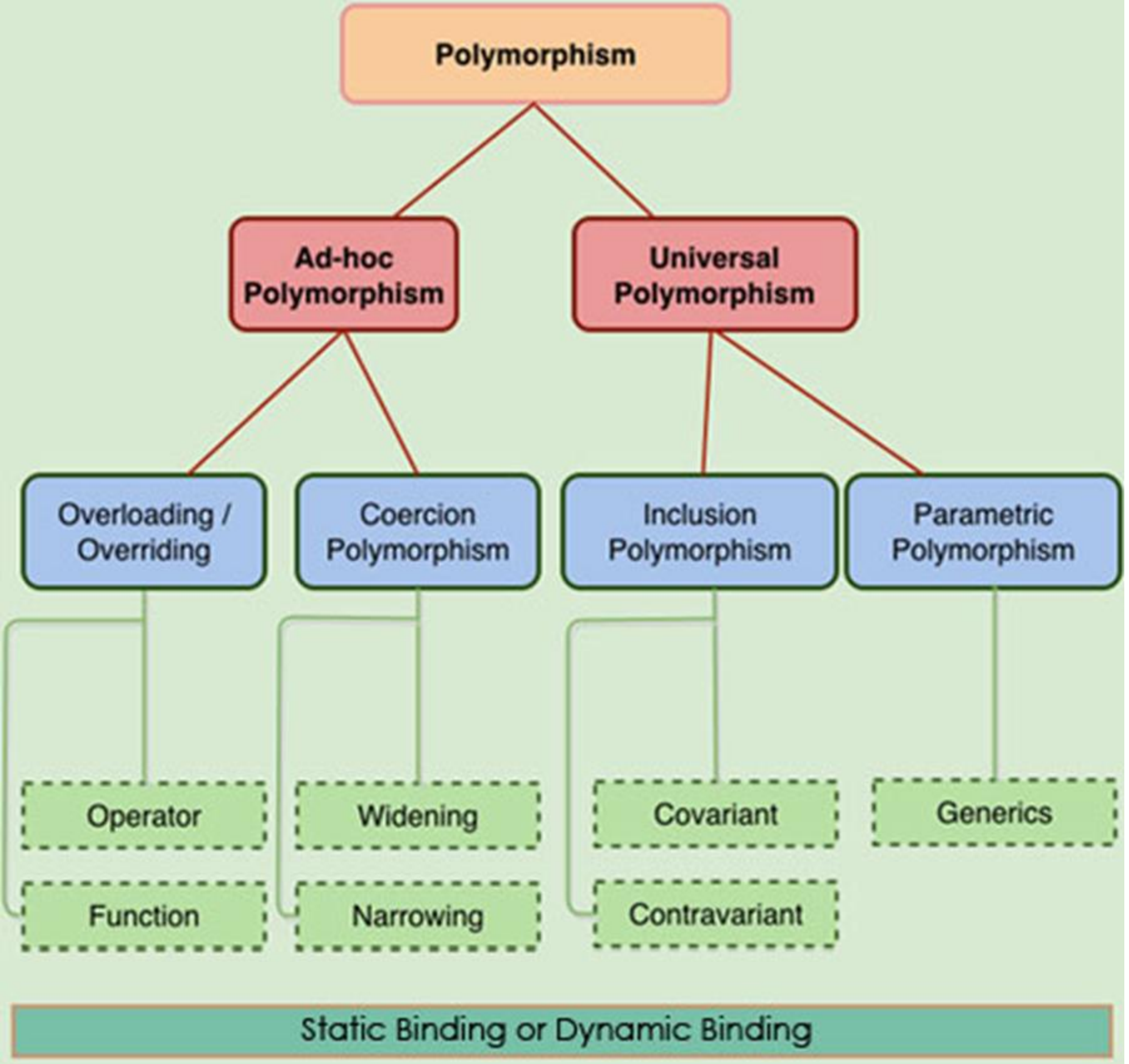
To be done.

Polymorphism

LECTURE 1

Polymorphism

- "Functions that can work with several types are called **polymorphic.**" – Downey, *Think Python*
- "The primary usage of **polymorphism** in industry (object-oriented programming theory) is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior.
- The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run time (this is called **late binding** or **dynamic binding**)." - *Wikipedia*



General Polymorphism Topics (Not Python-Specific)

Types of Polymorphism: (By assignment time)

- **Ad hoc Polymorphism:** polymorphism assigned by programmer at any time in design time.
- **Universal Polymorphism:** polymorphism assigned by language definition.

Types of Polymorphism: (By purpose)

- **Overloading/Overriding:** Re-definition of functions
- **Coercion Polymorphism:** Data Casting
- **Inclusion Polymorphism:** Sub-type polymorphism or polymorphism by inheritance
- **Parametric Polymorphism:** Generics, Generic data type

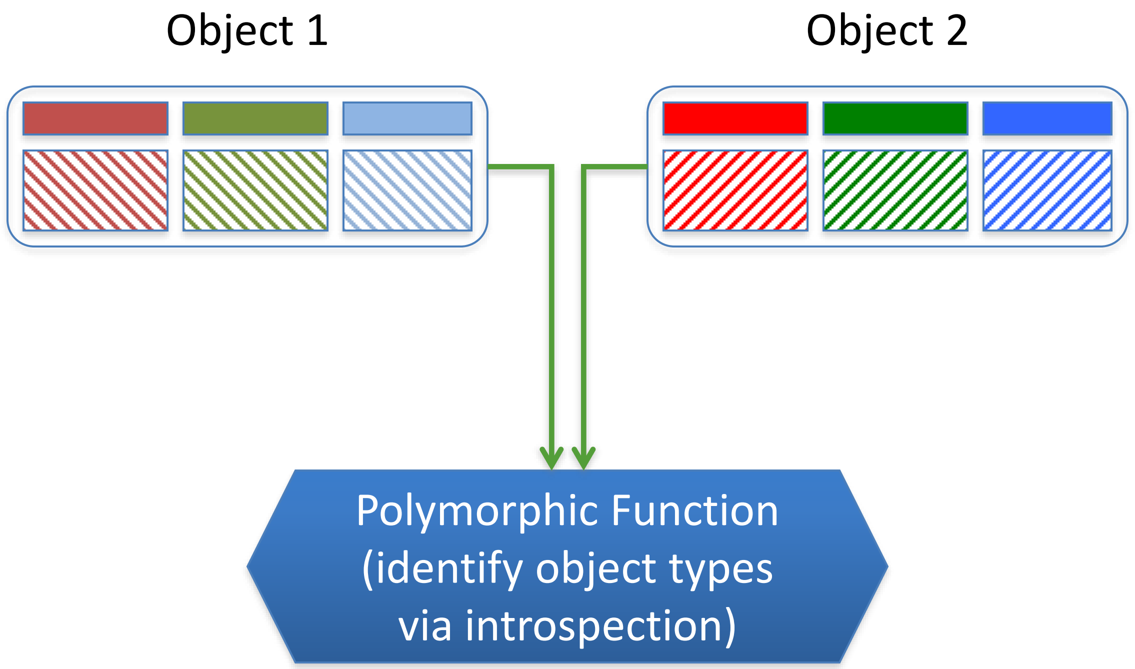
Python Polymorphism

- Python does have the need for Coercion and Parametric Polymorphism.
- Python has default inclusion Polymorphism
- Python accepts Overloading Polymorphism

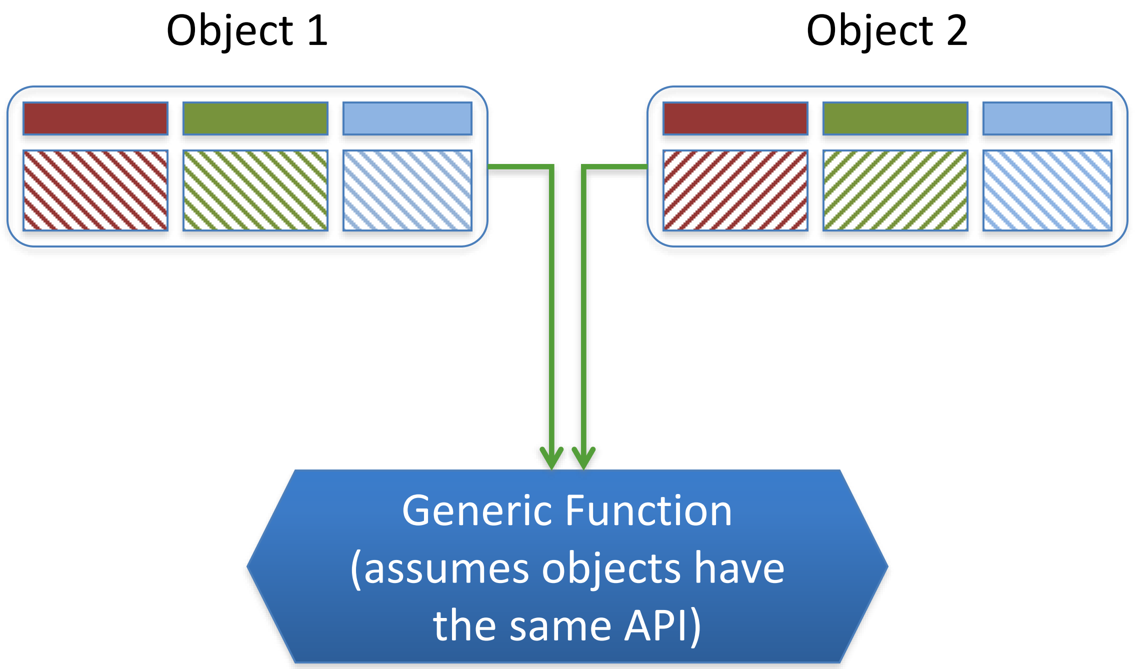
Polymorphism

- Generic Containers (Generic Data Structure)
- Generic Methods
- Generic Parameters (Python’s Parameters are all generic – subclass of objects)

Polymorphic Function



Polymorphic Classes



Polymorphism

The critical feature of polymorphism is a shared **interface**

- Using the Downey definition, we present a common interface where the same function may be used regardless of the argument type
- Using the Wikipedia definition, we require that polymorphic objects share a common interface that may be used to manipulate the objects regardless of type (class)

Polymorphism

- Why is polymorphism useful?
  - By reusing the same interface for multiple purposes, polymorphism reduces the number of "things" we have to remember
  - It becomes possible to write a "generic" function that perform a particular task, eg sorting, for many different classes (instead of one function for each class)

Polymorphism

- To define a polymorphic function that accepts multiple types of data requires the function either:
  - be able to distinguish among the different types that it should handle, or
  - be able to use other polymorphic functions, methods or syntax to manipulate any of the given types

Polymorphic Method: Type-based Dispatch

Demo Program: dispatch.py

Python provides several ways of identifying data types:

- **isinstance** function
- **hasattr** function
- **__class__** attribute

```
def what_is_this(data):
    if isinstance(data, str):
            return "instance of string"
    elif hasattr(data, "__class__"):
            return ("instance of %s" %  data.__class__.__name__)
    raise TypeError("unknown type: %s" % str(data))

class NC(object): pass
class OC: pass
print(what_is_this("Hello"))
print(what_is_this(12))
print(what_is_this([1, 2]))
print(what_is_this({12:14}))
print(what_is_this(NC()))
print(what_is_this(OC()))
```

```
   object
     |
     |
 basestring
    / \
   /   \
 str   unicode
```

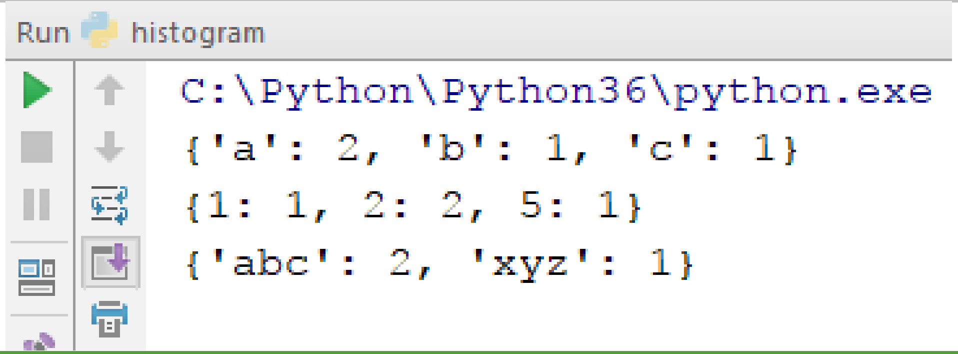
Python 3: basestring is replaced by str

Polymorphic Syntax

Demo Program: histogram.py (Polymorphic Method)

- Python uses the same syntax for a number of data types, so we can implement polymorphic functions for these data types if we use the right syntax
- Python data types are all subclass of object. Therefore, it is considered to be polymorphic by its language nature.

```
def histogram(s):
    d = dict()
    for c in s:
        d[c] = d.get(c, 0) + 1
    return d
print histogram("aabc")
print histogram([1, 2, 2, 5])
print histogram(("abc", "abc", "xyz"))
```



```
C:\Python\Python36\python.exe
{'a': 2, 'b': 1, 'c': 1}
{1: 1, 2: 2, 5: 1}
{'abc': 2, 'xyz': 1}
```

Polymorphic Classes

Classes that share a common interface

- A function implemented using only the common interface will **work with objects from any of the classes**

Although Python does not require it, a simple way to achieve this is to have the classes derive from a common superclass

- To maintain polymorphism, methods overridden in the subclasses ***must*** keep the same arguments as the method in the superclass

Polymorphic Classes

- The superclass defining the interface often has no implementation and is called an **abstract base class**
- Subclasses of the abstract base class override interface methods to provide class-specific behavior
- A generic function can manipulate all subclasses of the abstract base class

Polymorphic Classes

Demo Program: Series.py

In our example, all three subclasses overrode the **next** method of the base class, so they each have different behavior

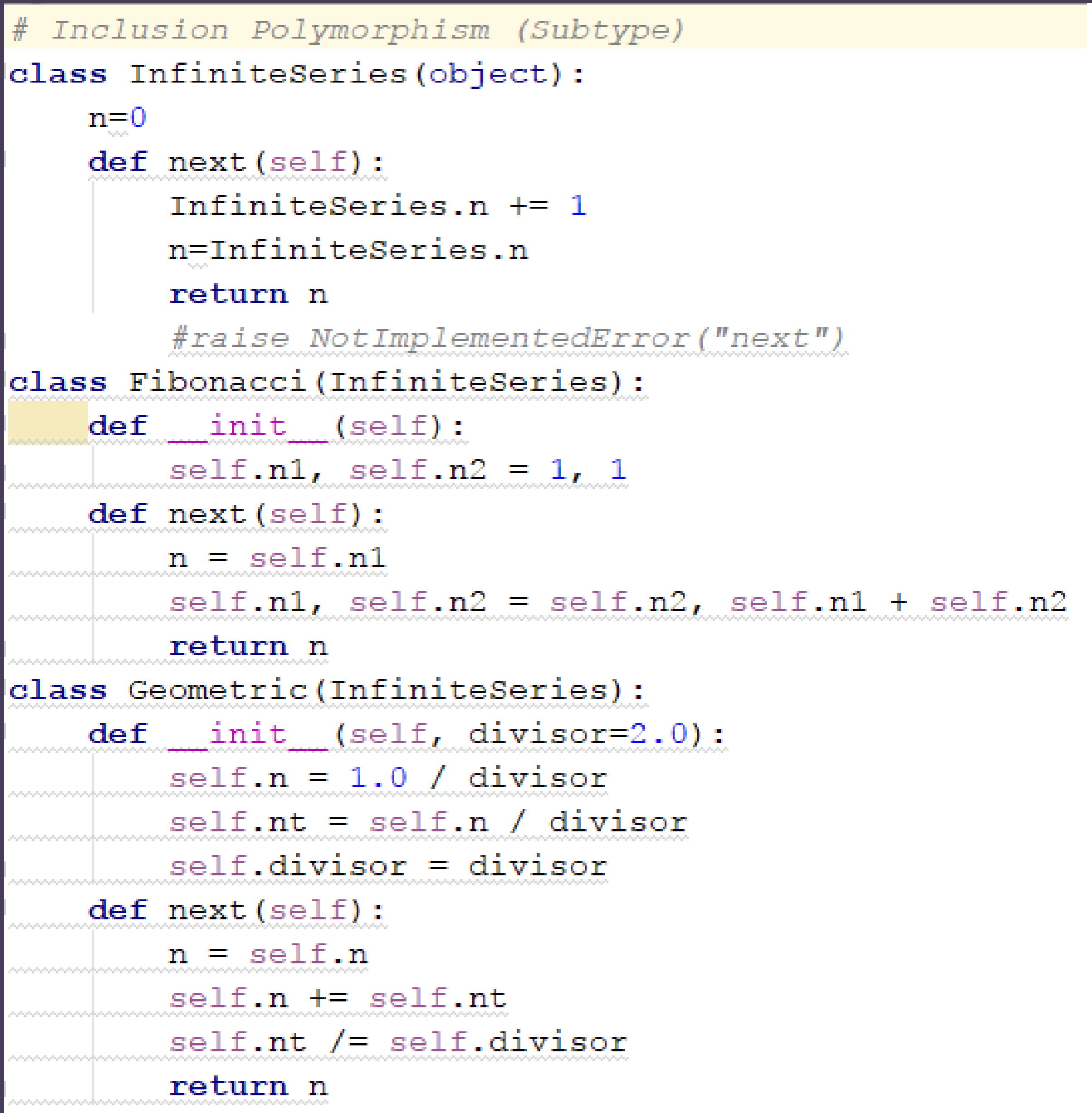
If a subclass does ***not*** override a base class method, then it **inherits** the base class behavior

- If the base class behavior is acceptable, the writer of the subclass does not need to do anything
- There is only one copy of the code so, when a bug is found it the inherited method, only the base class needs to be fixed

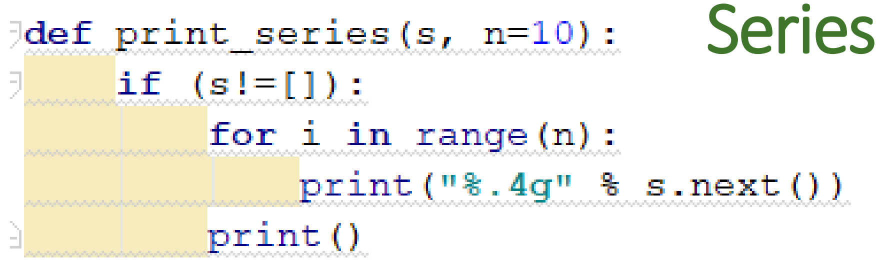
instance.method() is preferable over ***class.method(instance)***

- Although the code still works, the explicit naming of a class in the statement suggests that the method is defined in the class when it might actually be inherited from a base class

Series.py



```python
# Inclusion Polymorphism (Subtype)
class InfiniteSeries(object):
    n=0
    def next(self):
        InfiniteSeries.n += 1
        n=InfiniteSeries.n
        return n
        #raise NotImplementedError("next")
class Fibonacci(InfiniteSeries):
    def __init__(self):
        self.n1, self.n2 = 1, 1
    def next(self):
        n = self.n1
        self.n1, self.n2 = self.n2, self.n1 + self.n2
        return n
class Geometric(InfiniteSeries):
    def __init__(self, divisor=2.0):
        self.n = 1.0 / divisor
        self.nt = self.n / divisor
        self.divisor = divisor
    def next(self):
        n = self.n
        self.n += self.nt
        self.nt /= self.divisor
        return n
```



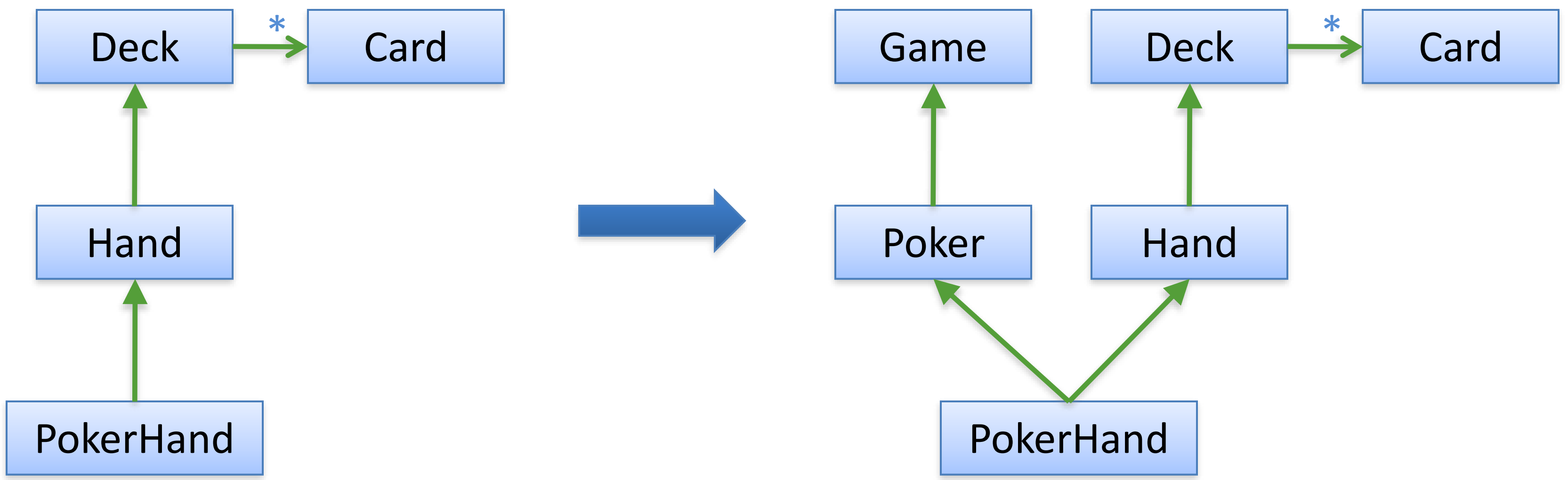
```python
def print_series(s, n=10):
    if (s!=[]):
        for i in range(n):
            print("%.4g" % s.next())
        print()

print("Fibonacci: ")
print_series(Fibonacci())
print("Geometric: ")
print_series(Geometric(3.0))
print("Infinite:  ")
print_series(InfiniteSeries())
```

Output:

```
Fibonacci:   Geometric:   Infinite:
1            0.3333       1
1            0.4444       2
2            0.4815       3
3            0.4938       4
5            0.4979       5
8            0.4993       6
13           0.4998       7
21           0.4999       8
34           0.5          9
55           0.5          10
```

Cards, *Decks* and *Hands*

Class diagram of example in Chapter 18 and Exercise 18.6



Is More Complex Better?

Advantages

- Each class corresponds to a real concept
- It should be possible to write a polymorphic function to play cards using only Game and Hand interfaces
- It should be easier to implement other card games

Disadvantages

- More classes means more things to remember
- Need multiple inheritance (although in this case it should not be an issue because the class hierarchy is simple)

Introspection

LECTURE 1

What is introspection?

- In everyday life, introspection is the act of self-examination. Introspection refers to the examination of one's own thoughts, feelings, motivations, and actions.
- Python's support for introspection runs deep and wide throughout the language.
- In fact, it would be hard to imagine Python without its introspection features. By the end of this article you should be very comfortable poking inside the hearts and souls of your own Python objects.

sys.platform

sys.version

sys.maxsize

sys.maxint not in Python3

sys.argv

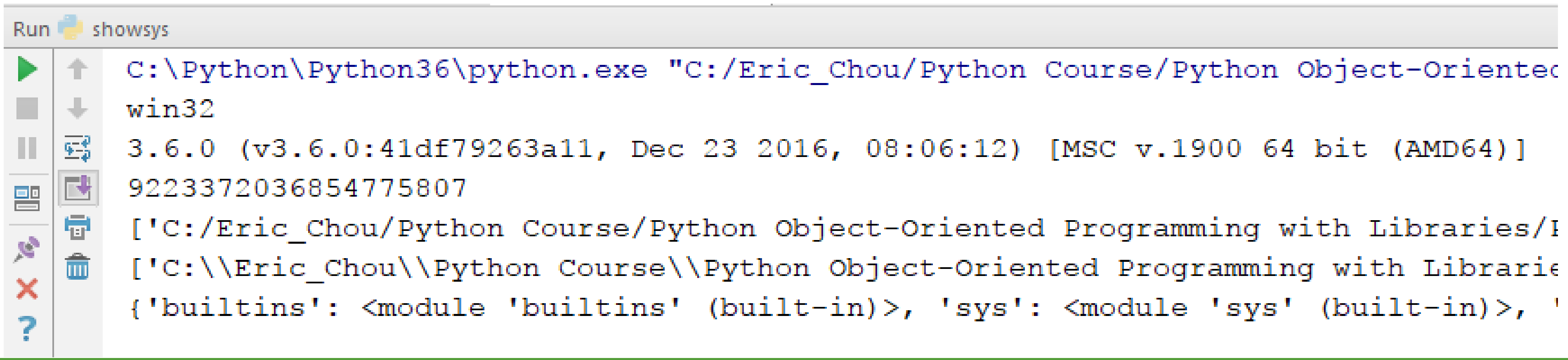
sys.path

sys.modules

The sys module

Demo Program: showsys.py

Go PyCharm!!!



```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python Object-Orientec
win32
3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)]
9223372036854775807
['C:/Eric_Chou/Python Course/Python Object-Oriented Programming with Libraries/I
['C:\\Eric_Chou\\Python Course\\Python Object-Oriented Programming with Librarie
{'builtins': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, '
```

The dir() function

- While it's relatively easy to find and import a module, it isn't as easy to remember what each module contains. And you don't always want to have to look at the source code to find out.
- Fortunately, Python provides a way to examine the contents of modules (and other objects) using the built-in dir() function.
- The dir() function is probably the **most well-known of all of Python's introspection mechanisms**.
- It returns a sorted list of attribute names for any object passed to it. If no object is specified, dir() returns the names in the current scope.

dir() function

Searching for the Attributes and Environment Parameters.

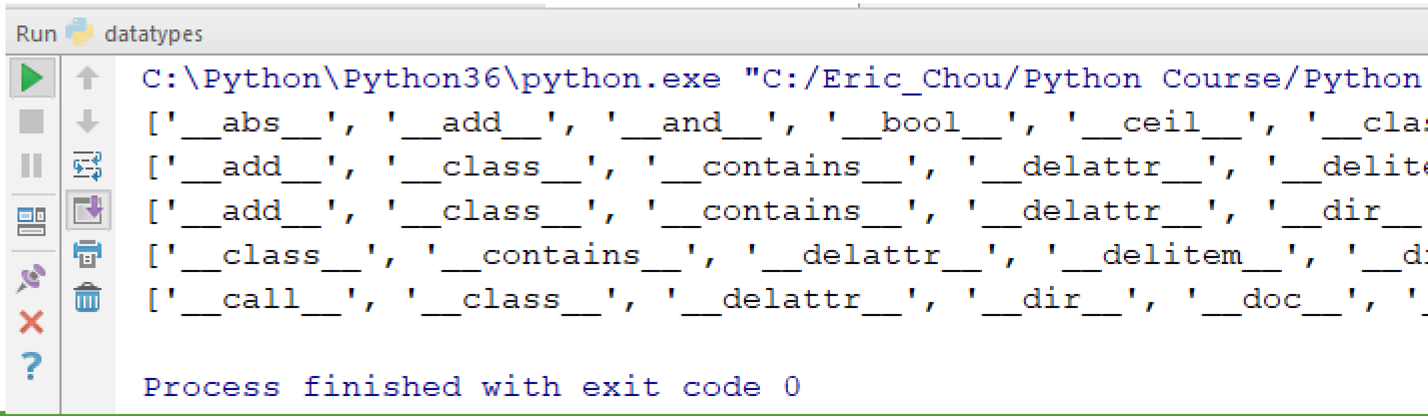
- **dir(keyword)**: The keyword module's attributes
- **dir(sys)**: The sys module's attributes
- **dir()**: Names in the current scope
- **__builtins__**: __builtins__ appears to be a name in the current scope that's bound to the module object named __builtin__.
- **dir(__builtins__)**: The __builtins__ module's attributes.
- **dir('this is a string')**: String attributes

Checking Attributes for Data Types

Demo Program: datatype.py

Go PyCharm!!!

```
print(dir(42))    # Integer (anprint(d the meaning of life)
print(dir([]))    # List (an empty list, actually)
print(dir(()))    # Tuple (also empty)
print(dir({}))    # print(dictionary (print(ditto)
print(dir(dir))   # Function (functions are also objects)
```



Run datatypes

```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__cla
['__add__', '__class__', '__contains__', '__delattr__', '__delit
['__add__', '__class__', '__contains__', '__delattr__', '__dir__
['__class__', '__contains__', '__delattr__', '__delitem__', '__d
['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '

Process finished with exit code 0
```

```
>>> class Person(object):
...     """Person class."""
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def intro(self):
...         """Return an introduction."""
...         return "Hello, my name is %s and I'm %s." % (self.name, self.age)
...
>>> bob = Person("Robert", 35)   # Create a Person instance
>>> joe = Person("Joseph", 17)   # Create another
>>> joe.sport = "football"       # Assign a new attribute to one instance
>>> dir(Person)      # Attributes of the Person class
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'intro']
>>> dir(bob)         # Attributes of bob
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name']
>>> dir(joe)         # Note that joe has an additional attribute
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name', 'sport']
>>> bob.intro()      # Calling bob's intro method
"Hello, my name is Robert and I'm 35."
>>> dir(bob.intro)   # Attributes of the intro method
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__get__',
'__getattribute__', '__hash__', '__init__', '__new__', '__reduce__',
'__repr__', '__setattr__', '__str__', 'im_class', 'im_func', 'im_self']
```

Using dir() on Console in Debugging Mode

- To illustrate the dynamic nature of Python's introspection capabilities, let's look at some examples using dir() on a custom class and some class instances.
- We're going to define our own class interactively, create some instances of the class, add a unique attribute to only one of the instances, and see if Python can keep all of this straight.

Debugging

Python is capable of **introspection**, the ability to examine an object at run-time without knowing its class and attributes *a priori*

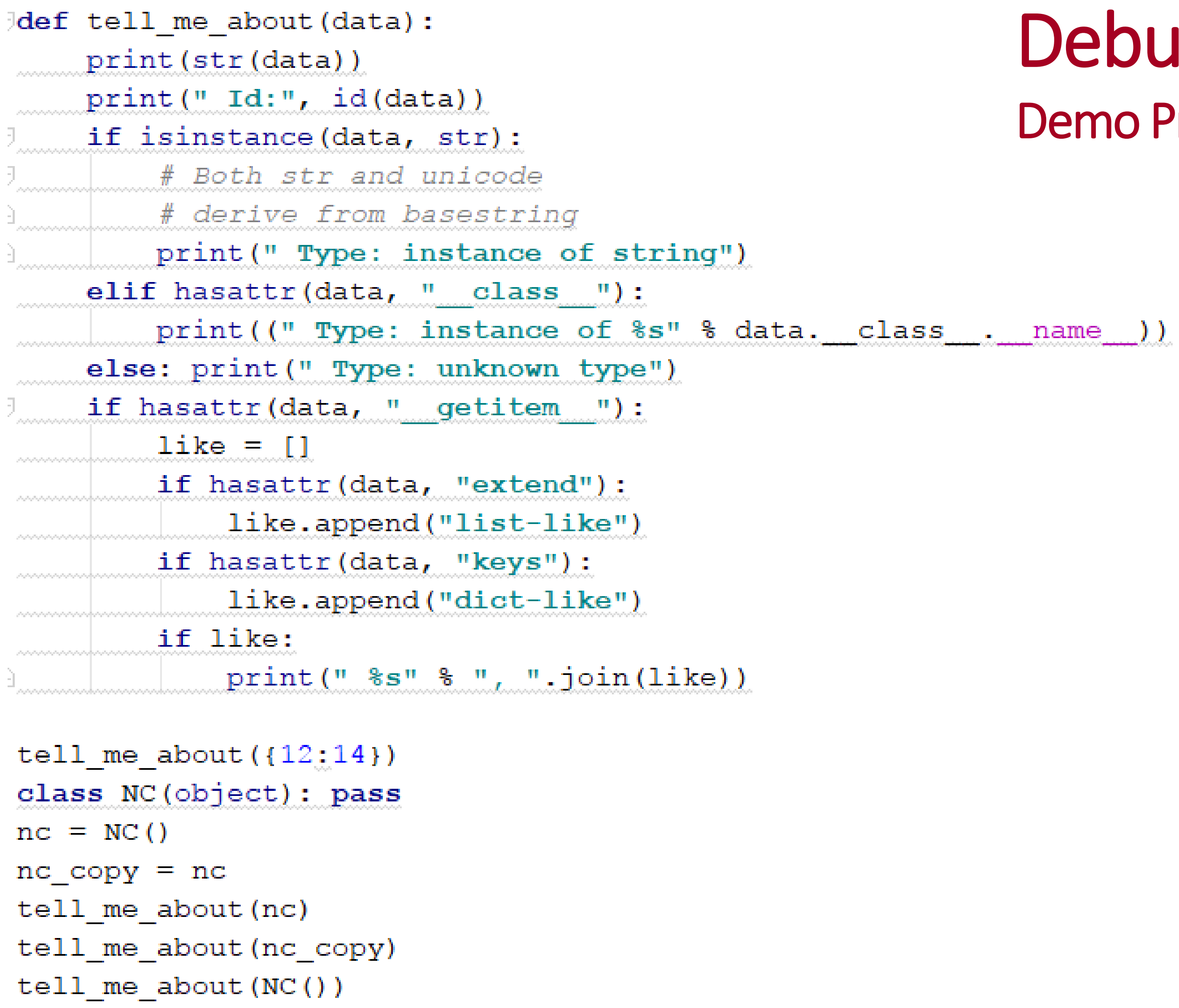
Given an object, you can

- get the names and values of its attributes (including inherited ones)
- get its class
- check if it is an instance of a class or a subclass of a class

Using these tools, you can collect a lot of debugging information using polymorphic functions

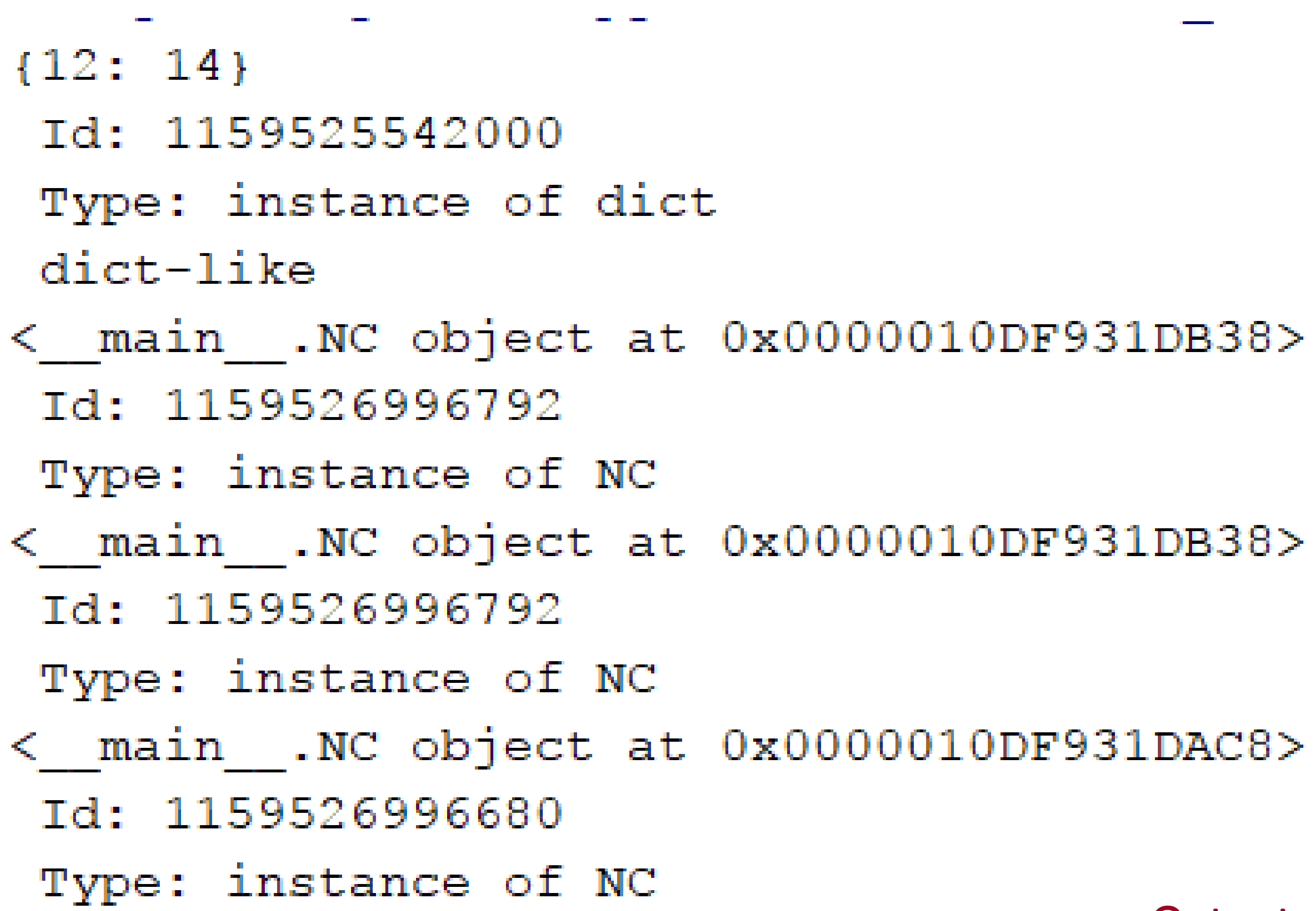
Debugging with Introspection

Demo Program: debugging.py



```
def tell_me_about(data):
    print(str(data))
    print(" Id:", id(data))
    if isinstance(data, str):
        # Both str and unicode
        # derive from basestring
        print(" Type: instance of string")
    elif hasattr(data, "__class__"):
        print((" Type: instance of %s" % data.__class__.__name__))
    else: print(" Type: unknown type")
    if hasattr(data, "__getitem__"):
        like = []
        if hasattr(data, "extend"):
            like.append("list-like")
        if hasattr(data, "keys"):
            like.append("dict-like")
        if like:
            print(" %s" % ", ".join(like))

tell_me_about({12:14})
class NC(object): pass
nc = NC()
nc_copy = nc
tell_me_about(nc)
tell_me_about(nc_copy)
tell_me_about(NC())
```

Introspection is a way for programmer to dump some environment and/or object information to help debugging.



```
{12: 14}
 Id: 1159525542000
 Type: instance of dict
 dict-like
<__main__.NC object at 0x0000010DF931DB38>
 Id: 1159526996792
 Type: instance of NC
<__main__.NC object at 0x0000010DF931DB38>
 Id: 1159526996792
 Type: instance of NC
<__main__.NC object at 0x0000010DF931DAC8>
 Id: 1159526996680
 Type: instance of NC
```

Output of debugging.py

More Introspection

Demo Program: showlist.py

```
def list_attributes(obj):
    for attr_name in dir(obj):
    print " %s:" % attr_name,
    value = getattr(obj, attr_name)
    if callable(value):
        print("function/method“)
    else:
        print(value)
list_attributes(list)
```

```
 __add__: <slot wrapper '__add__' of 'list' objects>
 __class__: <class 'type'>
 __contains__: <slot wrapper '__contains__' of 'list' objects>
 __delattr__: <slot wrapper '__delattr__' of 'object' objects>
 __delitem__: <slot wrapper '__delitem__' of 'list' objects>
 __dir__: <method '__dir__' of 'object' objects>
 __doc__: list() -> new empty list
list(iterable) -> new list initialized from iterable's items
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
 __eq__: <slot wrapper '__eq__' of 'list' objects>
 __format__: <method '__format__' of 'object' objects>
 __ge__: <slot wrapper '__ge__' of 'list' objects>
 __getattribute__: <slot wrapper '__getattribute__' of 'list' objects>
 __getitem__: <method '__getitem__' of 'list' objects>
 __gt__: <slot wrapper '__gt__' of 'list' objects>
 __hash__: None
 __iadd__: <slot wrapper '__iadd__' of 'list' objects>
 __imul__: <slot wrapper '__imul__' of 'list' objects>
 __init__: <slot wrapper '__init__' of 'list' objects>
 __init_subclass__: <built-in method __init_subclass__ of type object at 0x0000000073DAF530>
 __iter__: <slot wrapper '__iter__' of 'list' objects>
 __le__: <slot wrapper '__le__' of 'list' objects>
 __len__: <slot wrapper '__len__' of 'list' objects>
 __lt__: <slot wrapper '__lt__' of 'list' objects>
 __mul__: <slot wrapper '__mul__' of 'list' objects>
 __ne__: <slot wrapper '__ne__' of 'list' objects>
```

```
__ne__: <slot wrapper '__ne__' of 'list' objects>
__new__: <built-in method __new__ of type object at 0x0000000073DAF530>
__reduce__: <method '__reduce__' of 'object' objects>
__reduce_ex__: <method '__reduce_ex__' of 'object' objects>
__repr__: <slot wrapper '__repr__' of 'list' objects>
__reversed__: <method '__reversed__' of 'list' objects>
__rmul__: <slot wrapper '__rmul__' of 'list' objects>
__setattr__: <slot wrapper '__setattr__' of 'object' objects>
__setitem__: <slot wrapper '__setitem__' of 'list' objects>
__sizeof__: <method '__sizeof__' of 'list' objects>
__str__: <slot wrapper '__str__' of 'object' objects>
__subclasshook__: <built-in method __subclasshook__ of type object at 0x0000000073DAF530>
append: <method 'append' of 'list' objects>
clear: <method 'clear' of 'list' objects>
copy: <method 'copy' of 'list' objects>
count: <method 'count' of 'list' objects>
extend: <method 'extend' of 'list' objects>
index: <method 'index' of 'list' objects>
insert: <method 'insert' of 'list' objects>
pop: <method 'pop' of 'list' objects>
remove: <method 'remove' of 'list' objects>
reverse: <method 'reverse' of 'list' objects>
sort: <method 'sort' of 'list' objects>
```