



CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 5B Object-Oriented Programming Class-to-class Relationship

LECTURE 6: CLASS TO CLASS RELATIONSHIP

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Class to Class Relationship
- Encapsulation
- Inheritance
- Polymorphism
- Interface and Abstract Class
- Functional Closure

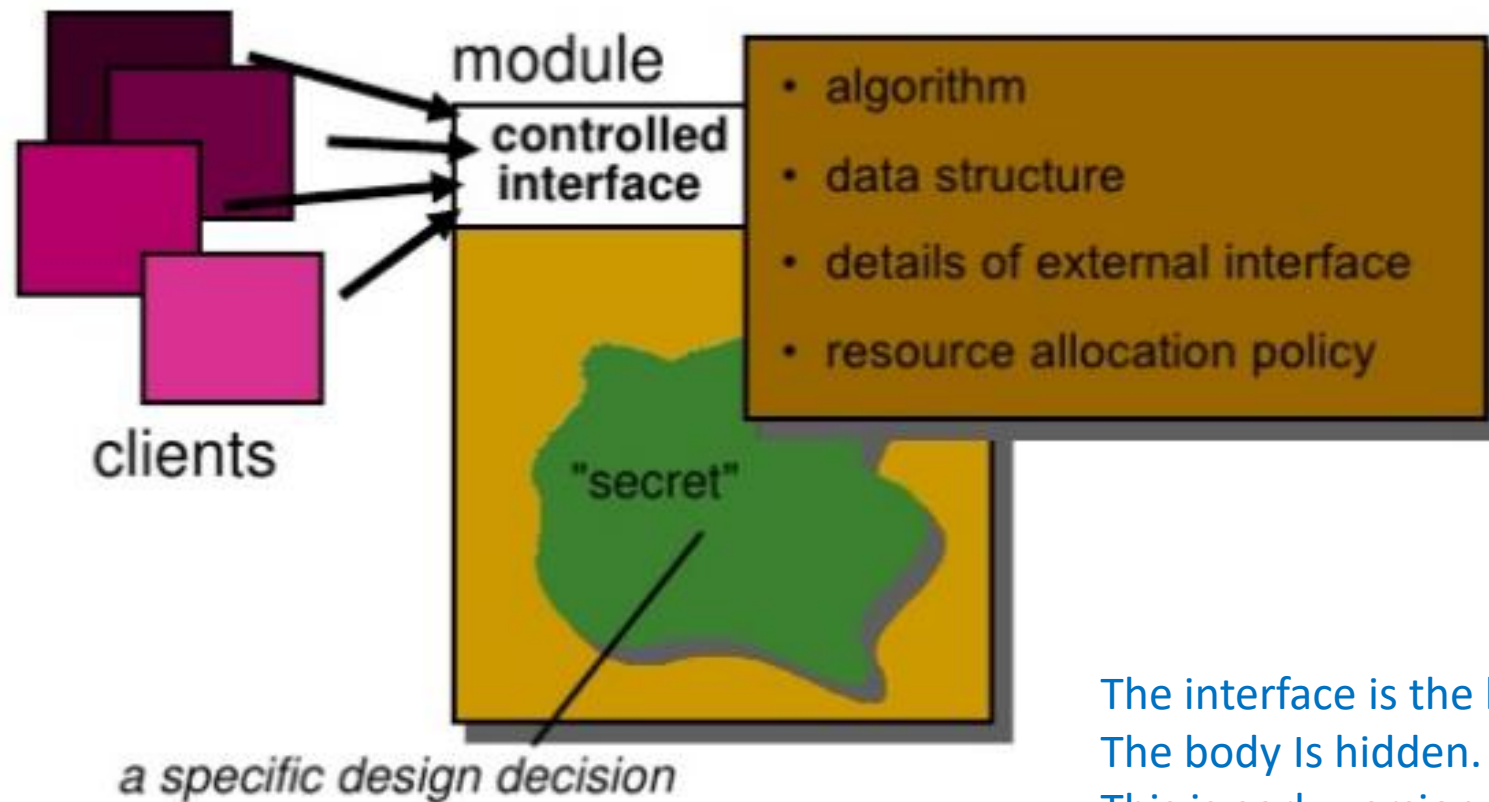
Objectives

- Python does not have interface. Duck function and Type Inference are used in lieu of interfaces.
- Understand what duck typing is.
- Try to use Python interface and function closure for advanced programming

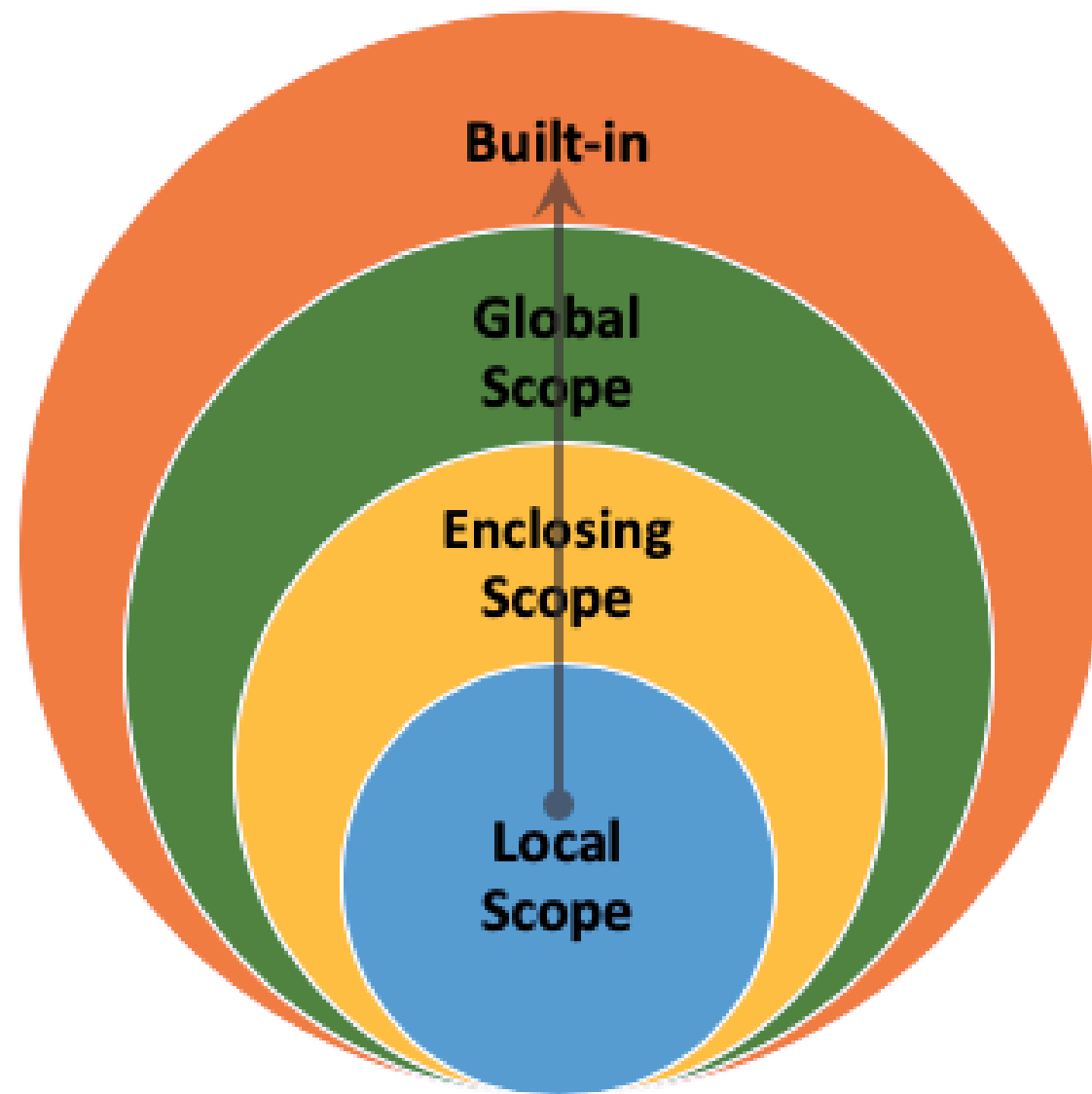
Encapsulation

SECTION 1

Information Hiding by Module



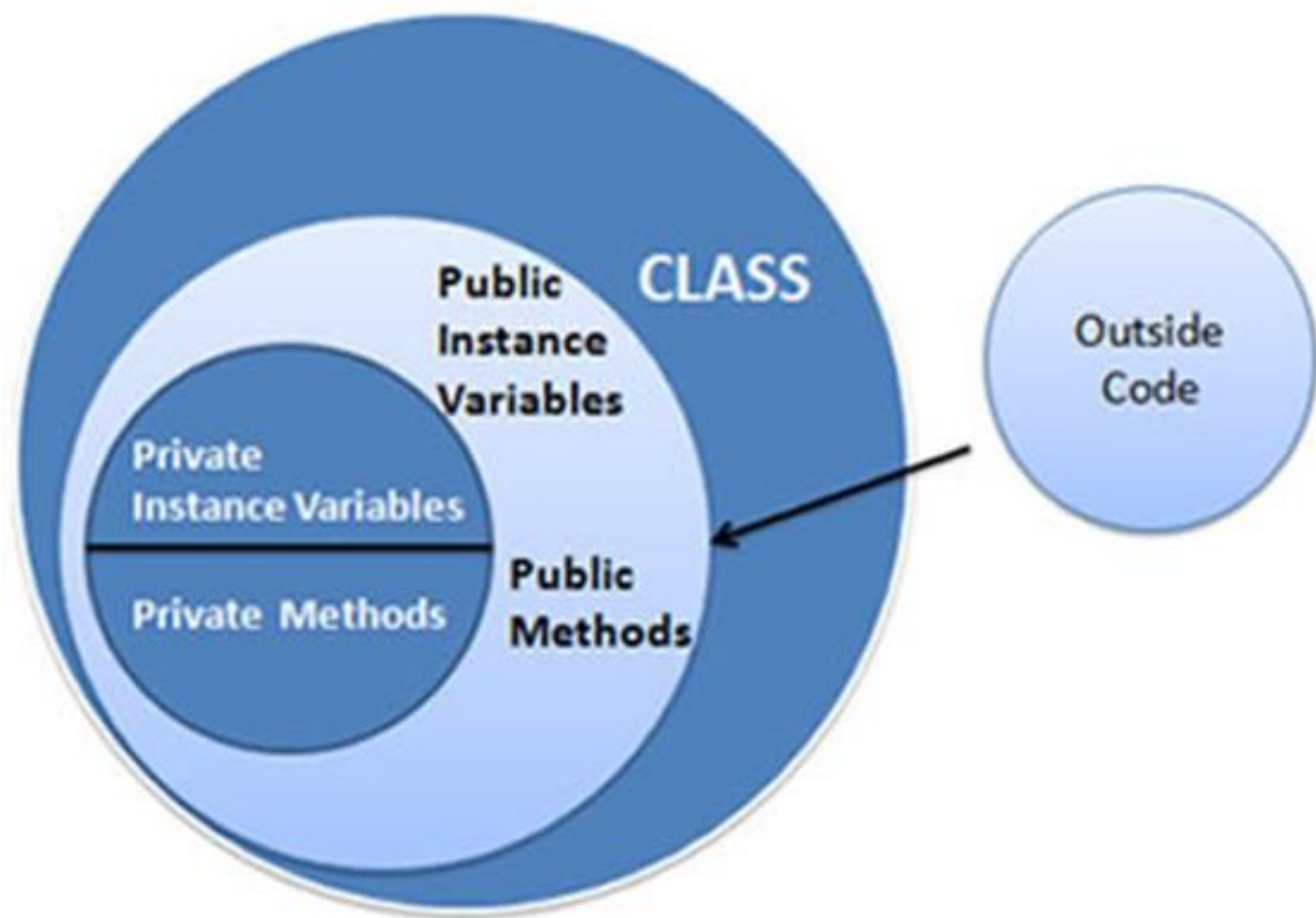
The interface is the header.
The body is hidden.
This is early version of data encapsulation.



Encapsulation

Information Hiding

- Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction.
- Topics for Encapsulation:
 - Data-hiding mechanisms of modules in non-object-oriented languages.
 - The new data-hiding issues, in OOP, that arise when we add inheritance to modules to make classes.
 - Module-as-manager approach, and show how several languages, including Ada 95 and Fortran 2003, add inheritance to records, allowing (static) modules to continue to provide data hiding.



Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
__name	Private	Can't be seen and accessed from outside

Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

finxter

Public Attribute In Python

- In Python, every member of the class is public by default.
- Public members in a class can be accessed from anywhere outside the class.
- You can access the public members by creating the object of the class.

```
class Music:
    # Creating a constructor
    def __init__(self):
        self.genre = "Pop"
        self.singer = "Coldplay"
        # These are public attributes
    # Creating a function
    def foo(self):
        song = 'Hymn For The Weekend'
        return song

# Creating object of the class
m = Music()

# Accessing the members inside the class
print("Song: ", m.foo())
print("Genre:", m.genre)
print("Singer:", m.singer)
```

Song: Hymn For The Weekend
Genre: Pop
Singer: Coldplay

Private Attributes in Python

- Unfortunately, Python does not have a way to effectively restrict access to instance variables or methods. ?
- However, we do have a workaround. To declare a member as private in Python, you have to use double underscore `__` as a prefix to the variables. Private members are restricted within the same class, i.e. we can access the private members only within the same class.

```
class Music2:
    # constructor
    def __init__(self):
        # These are private variables
        self.__genre = "Pop"
        self.__singer = "Coldplay"

    # private function
    def __func(self):
        print('Music: Hym For The Weekend')

    def foo(self):
        # Accessing private members of the class
        obj.__func()
        print("Genre:", obj.__genre)
        print("Singer:", obj.__singer)

obj = Music2()  # Creating an object of the Music class
obj.foo()      # calling the private function
```

Music: Hym For The Weekend
Genre: Pop
Singer: Coldplay

Private Attributes in Python

- If you try to access the private member outside the class, you will get an `AttributeError`.

```
class Music3:
    # constructor
    def __init__(self):
        # These are private variables
        self.__genre = "Pop"
        self.__singer = "Coldplay"
    # private function
    def __func(self):
        print('Music: Hym For The Weekend')

    def foo(self):
        # Accessing private members of the class
        print("Genre:", obj.__genre)
        print("Singer:", obj.__singer)

# Creating object of the class
obj = Music3()

# Trying to access the private attributes from outside the class
obj.__func()
print("Genre:", obj.__genre)
print("Singer:", obj.__singer)
```

```
Traceback (most recent call last):
  File "c:\Eric_Chou\Lewis
University\CS 46K Structure and
Interpretation of Computer
Programs\PyDev\Lecture
6\Music3.py", line 23, in
<module>
    obj.__func()
AttributeError: 'Music3' object
has no attribute '__func'
```


Private Attributes in Python

- So, this brings us to the question – **How to access the private attributes from outside the class?** Is there a way??

When you use a double underscore (e.g.,

`__var`

), Python plays around with the name giving it properties of a private attribute. However, the variable can still be accessed from outside the class using its obfuscated name. Hence, it is not strictly private. This brings us to a very important concept in Python – **Name Mangling**. You can access the private attributes outside the class using name mangling.

Name Mangling in Python

- Name Mangling is a process in Python, where, if a method has, in any event, two underscores before the name, and at the most one underscore following the name, it gets replaced with `_ClassName` before it, for instance, `__method()` becomes `_ClassName__method()`. Since the name is changed internally by the interpreter, so we cannot access the variable utilizing its original name and that is how you can hide data in Python.
- Note: Name Mangling is essentially used to avoid overriding the methods for parent classes by inherited classes.

```
# Defining a class
class Music4:
    # Creating a constructor
    def __init__(self):
        # These are private attributes
        self.__genre = "Pop"
        self.__singer = "Coldplay"
        # This is a public attribute
        self.releaseyear = 2000
    # Creating a function
    def foo(self):
        print("Song: Trouble")

# Creating object of the class
obj = Music4()
# Calling the method inside the class
obj.foo()
# Accessing the private members outside the class using name mangling
print("Genre:", obj._Music4__genre)
print("Singer:", obj._Music4__singer)
# Accessing the public member normally
print("Year of release:", obj.releaseyear)
```

Song: Trouble
Genre: Pop
Singer: Coldplay
Year of release: 2000

Protected Attributes in Python

- You can access protected attributes of a class from within the class, and they can also be accessed by the sub-classes. This facilitates inheritance in Python.
- To make a variable **protected**, you have to add a **single underscore** (e.g.

_x

) as a prefix to it. To make it truly protected, you also have to use a **property decorator**.

```
# Defining a class
class Music5:
    def __init__(self):
        self._song = 'Trouble'

    @property
    def foo(self):
        return self._song

    @foo.setter
    def foo(self, new_song):
        # overloading foo
        self._song = new_song

obj = Music5()
print('Song - ', obj.foo)
obj.foo = 'Hym For The Weekend'
print('New Song - ', obj.foo)
```

Song - Trouble
New Song - Hym For The Weekend

Explanation:

@property decorator ensured that `foo()` method is a property.

@foo.setter decorator allowed us to overload the `foo()` method.
Now, the variable `song` is protected. But this

? Alert! – You can still access `song` from outside using `obj._song`.
Thus you should always avoid accessing or modifying variables prefixed by `_` from outside the class.

Summary

- Thus, we learned about one of the most important OOP concepts in Python in this tutorial, i.e., how you can use public, private and protected attributes in Python.

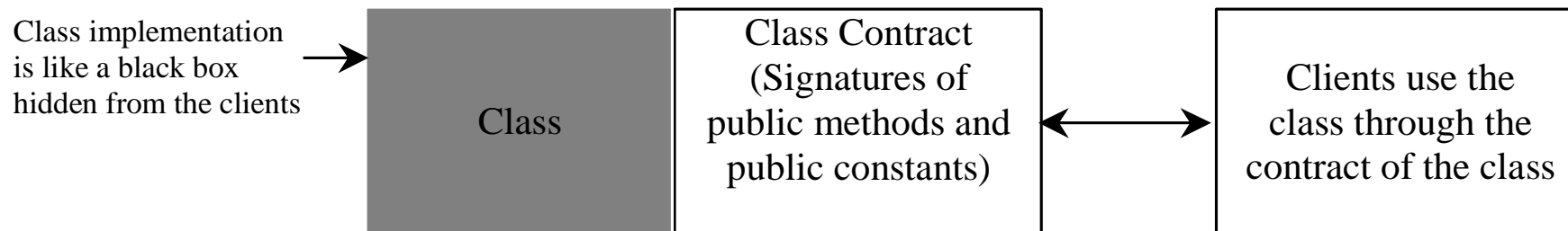
Classes and Objects

Class Relationship

SECTION 2

Class Abstraction and Encapsulation

Class abstraction means **to separate class implementation from the use of the class**. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Designing a Class: Coherence

- **(Coherence)** A class should describe a **single entity**, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

e.g. Student, Subject, ScoreSheet, Card, Deck, and Hand

Designing a Class, cont.

- **(Separating responsibilities)** A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.

Designing a Class, cont.

- Provide a public no-arg constructor and override the equals method and the toString method defined in the Object class whenever possible.
- Overriding standard methods inherited from object class.

Designing a Class, cont.

- Follow standard python programming style and naming conventions. Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.

Guidelines for Class Design

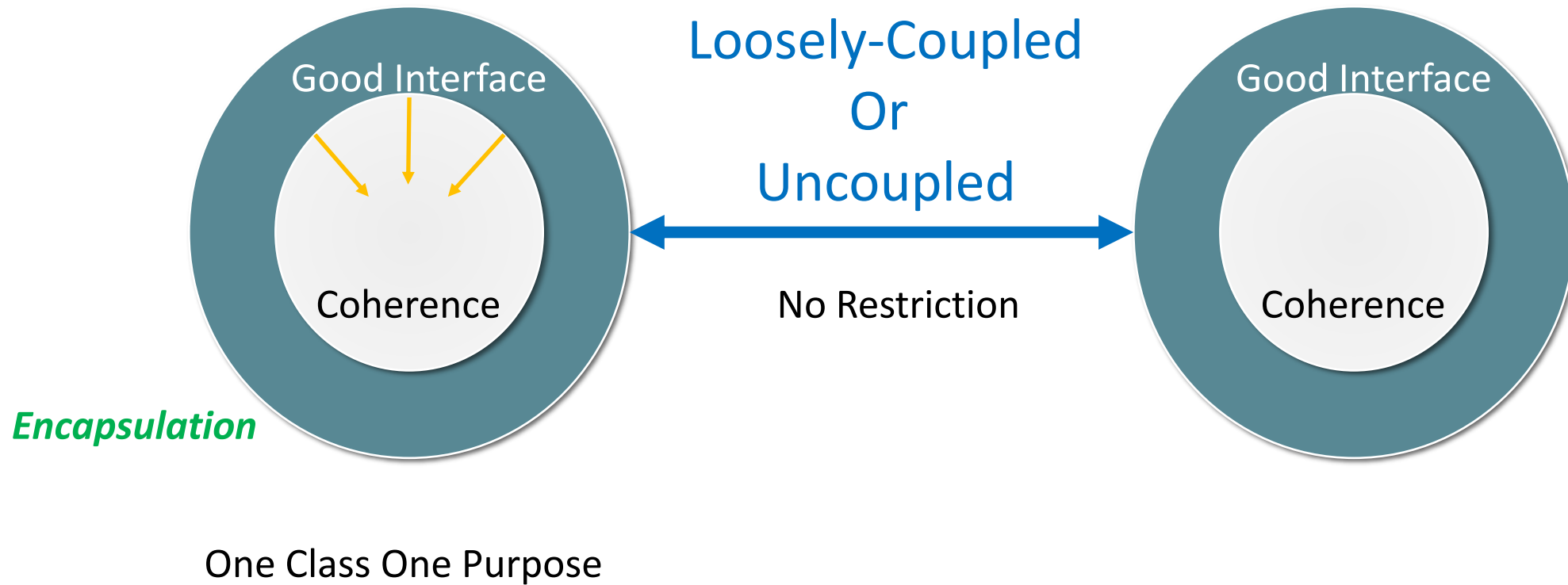
- Good design of individual classes is crucial to good overall system design. A well-designed class is **more re-usable** in different contexts, and **more modifiable** for future versions of software.
- Here, we'll look at some general class design guidelines, as well as some tips for specific languages, like **C++** or **Java**.

(Chapter 9's class design guidelines on technical issue, this chapter is on design styles)

General goals for building a good class

- Having a **good usable interface** (**Information Hiding**)
- **Implementation objectives**, like efficient algorithms and convenient/simple coding
- Separation of implementation from interface!! (**Encapsulation**)
- Improves **modifiability** and **maintainability** for future versions (**re-usability**)
- **Decreases coupling between classes**, i.e. the amount of dependency between classes (will changing one class require changes to another?) (**no restrictions**)
- Can re-work a class inside, without changing its interface, for outside interactions
- Consider how much the automobile has advanced technologically since its invention. Yet the basic interface remains the same -- steering wheel, gas pedal, brake pedal, etc.

Good Class Design

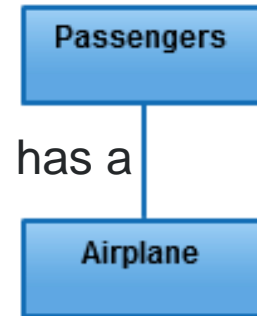
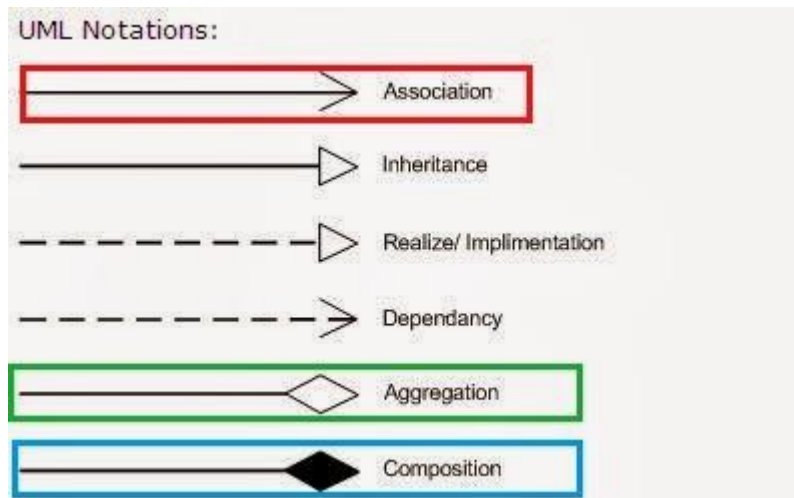
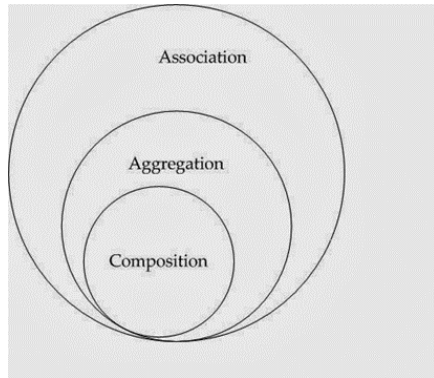


Designing a good class interface

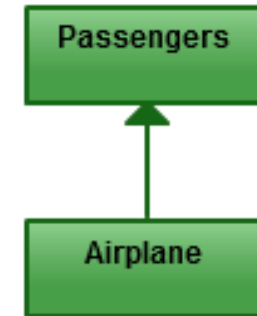
By interface, we are talking about what the class user sees. This is the public section of the class.

- **Cohesion** (or *coherence*): one class single abstraction
- **Completeness**: A class should support all important operations
- **Convenience**:
 1. User-friendly
 2. API-Oriented (written like API)
 3. Systematic
- **Clarity**: No confusion or ambiguous interpretations
- **Consistency**
 1. Operations in a class will be most clear if they are consistent with each other.
 2. Naming conventions (toString, compareTo, equals, isLetter, and etc.)

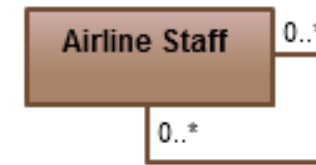
Types of Class-To-Class Relationships



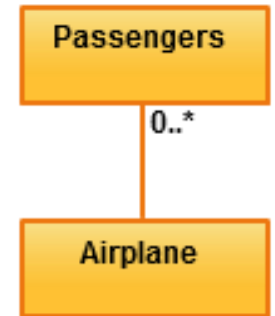
Association



Directed Association



Reflexive Association



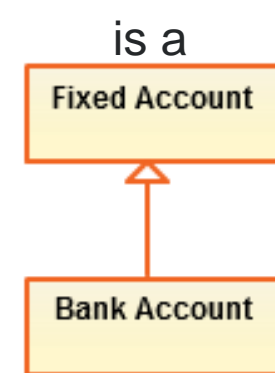
Multiplicity



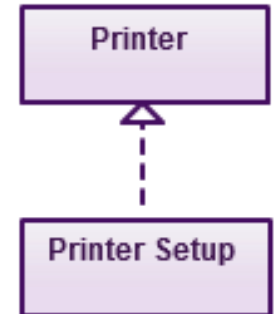
Aggregation



Composition



Inheritance



Realization

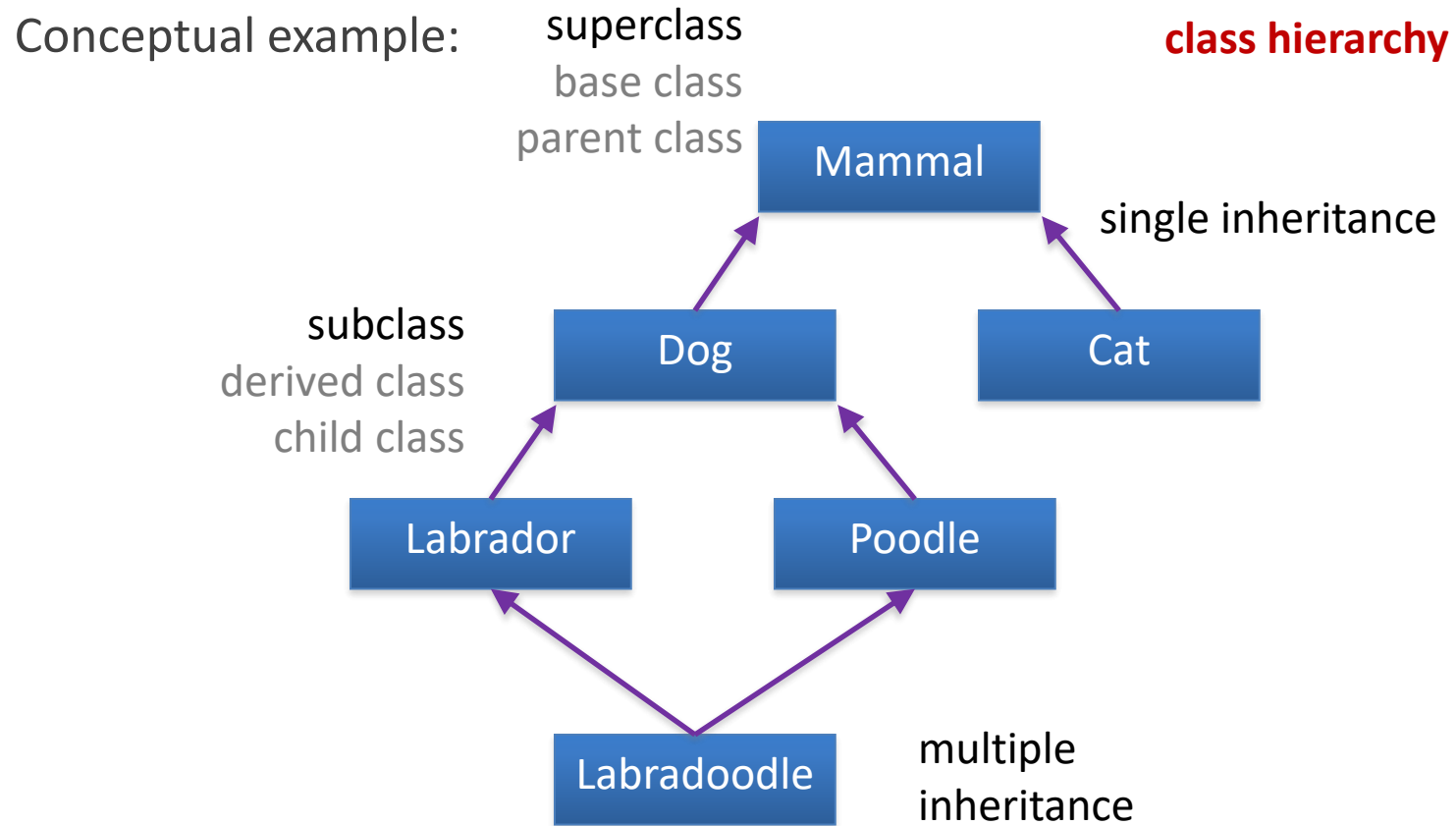
Inheritance

SECTION 3

Inheritance

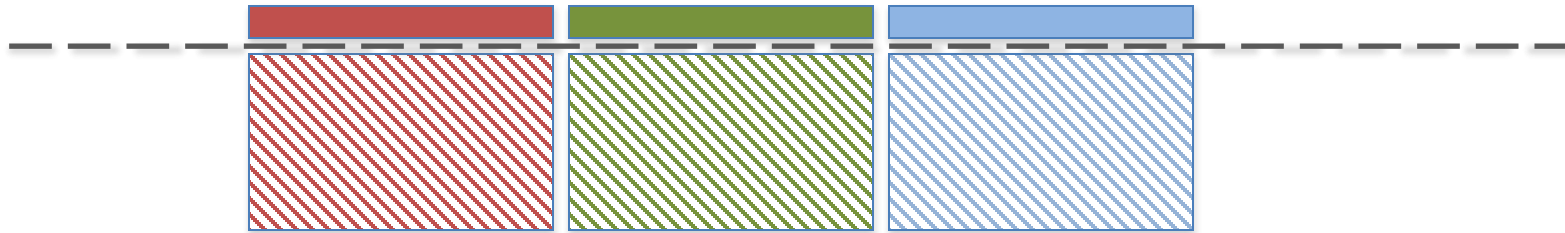
- Inheritance is the ability to define a new class that is a modified version of an existing class. – Allen Downey, *Think Python*
- “A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.
- Inheritance 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

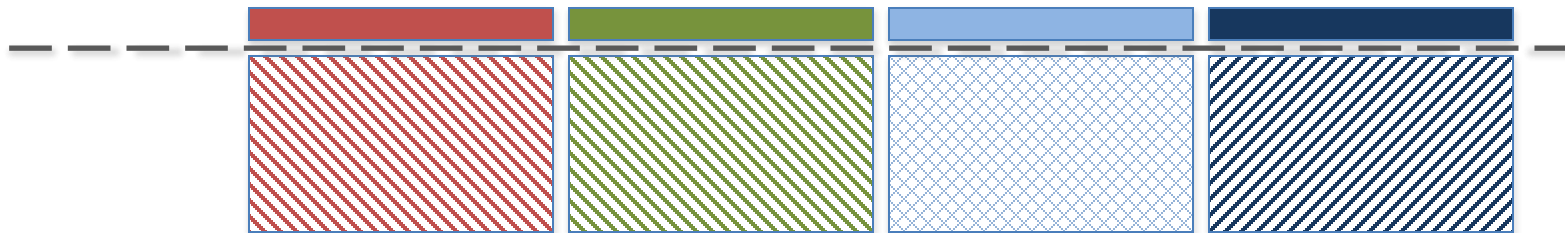


Base Class vs Derived Class

Base class



Derived class



Base Class

Feature 1

Feature 2

Derived Class

Feature 1

Feature 2

Feature 3

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

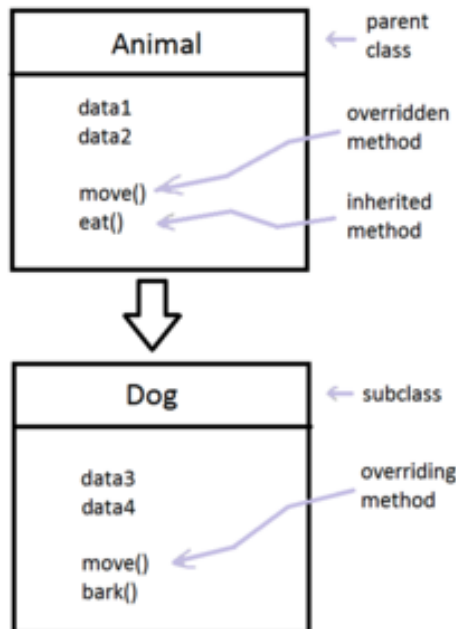
A subclass can **override** (replace or augment) methods of the superclass

- 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



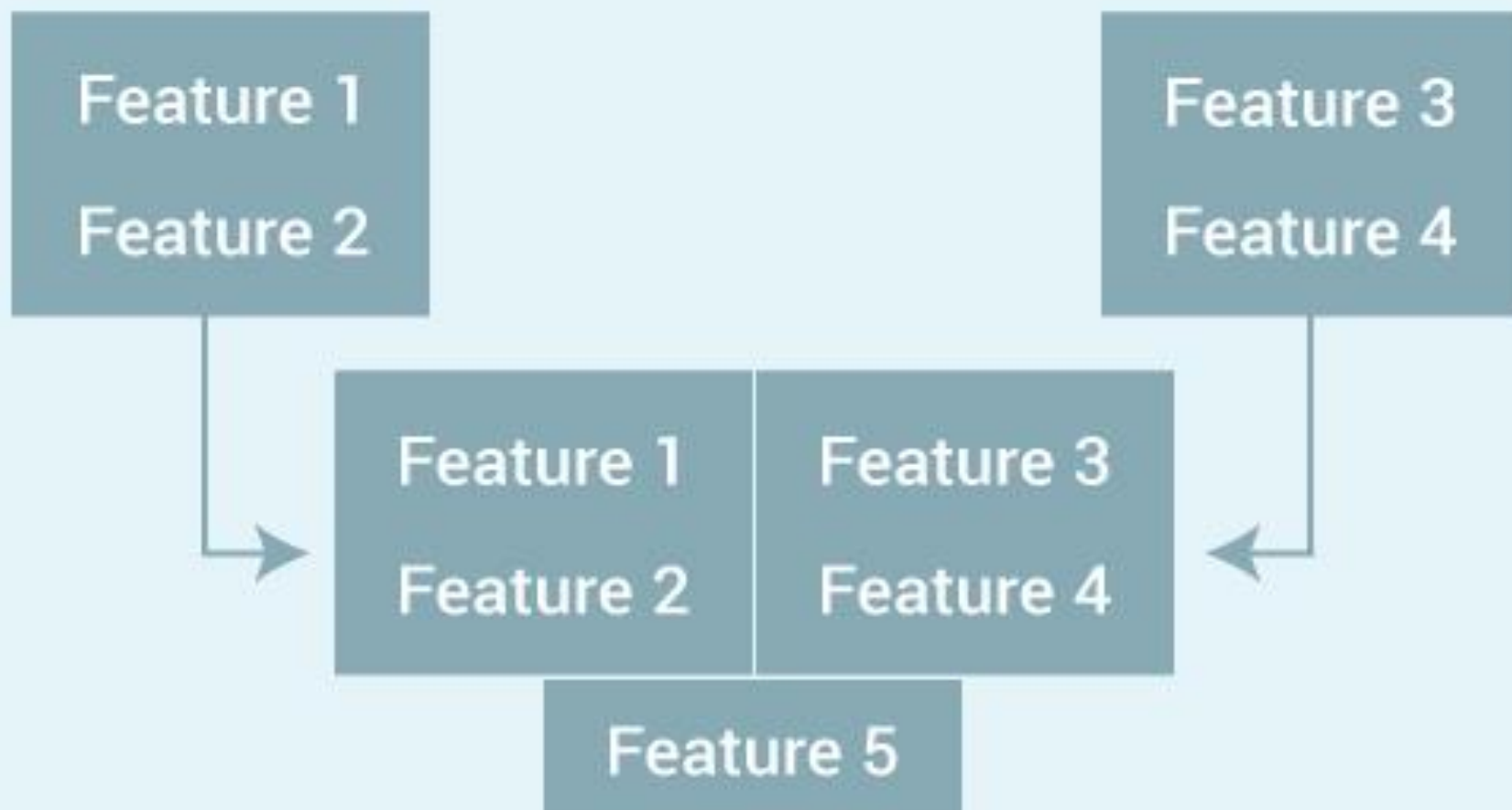
```
class Stack(list):  
    push = list.append  
  
class Calculator(Stack):  
    def __init__(self):  
        Stack.__init__(self)  
        self.accumulator = 0  
    def __str__(self):  
        return str(self.accumulator)  
    def push(self, value):  
        Stack.push(self, value)  
     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

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

Mixins – Multiple Inheritance

SECTION 4

Python Multiple Inheritance



Multiple Inheritance

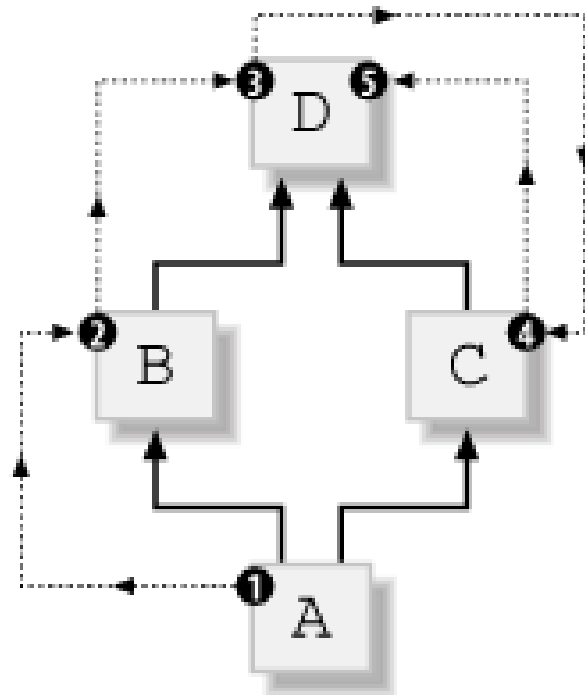
- Python supports multiple inheritance (This create chances to have collision in function names)
- In the **class** statement, replace the single superclass name with a comma-separated list of superclass names
- When looking up an attribute, Python will look for it in “Method Resolution Order” (MRO) 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 Algorithm

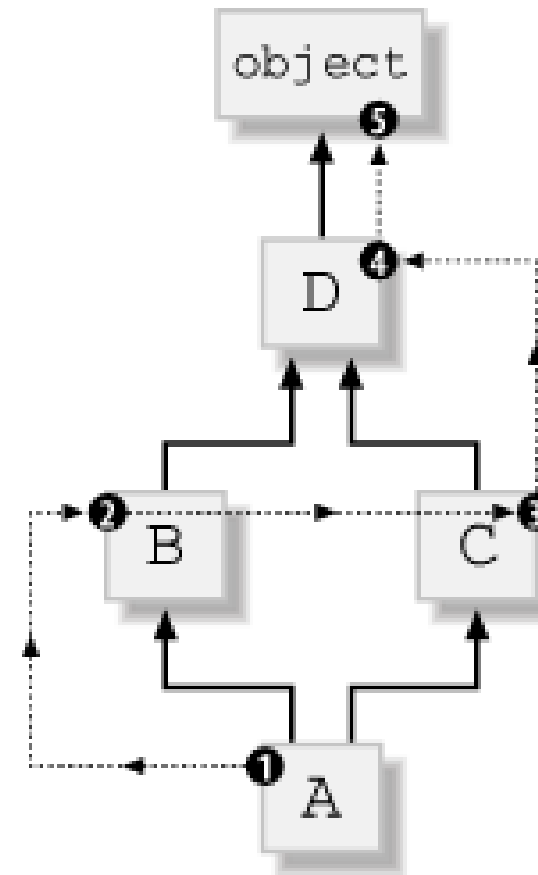
C3 Linearization

- 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


```
# python 3 MRO D-B-C-A
# swtich betwen pasa snd the functional definition to observe the MRO
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

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

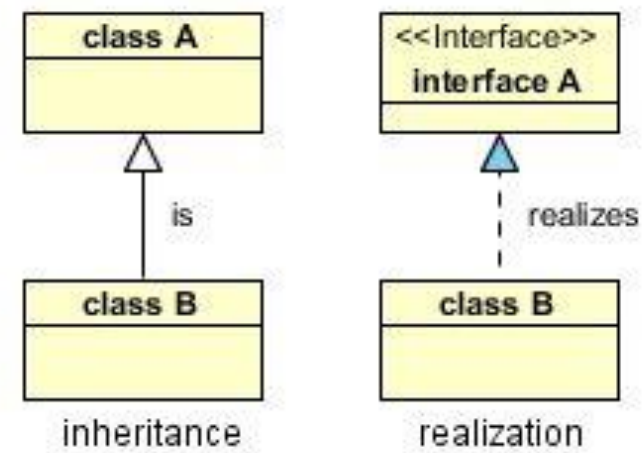
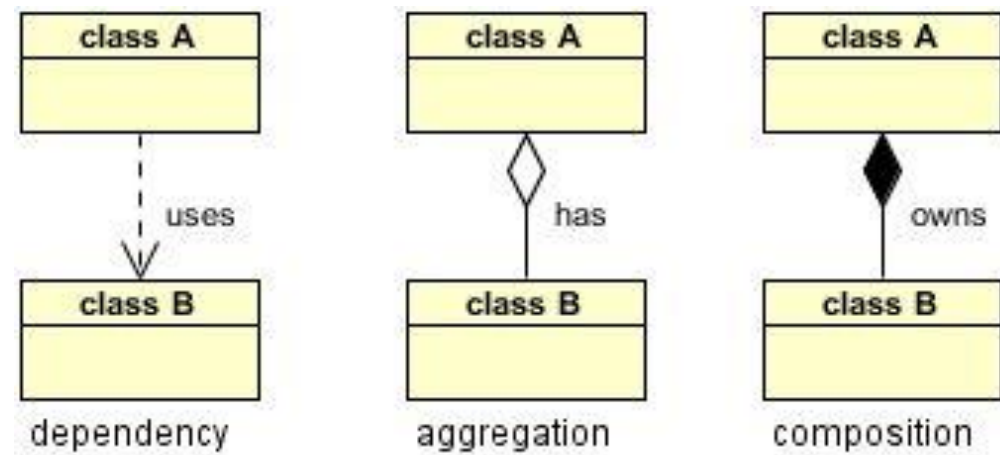
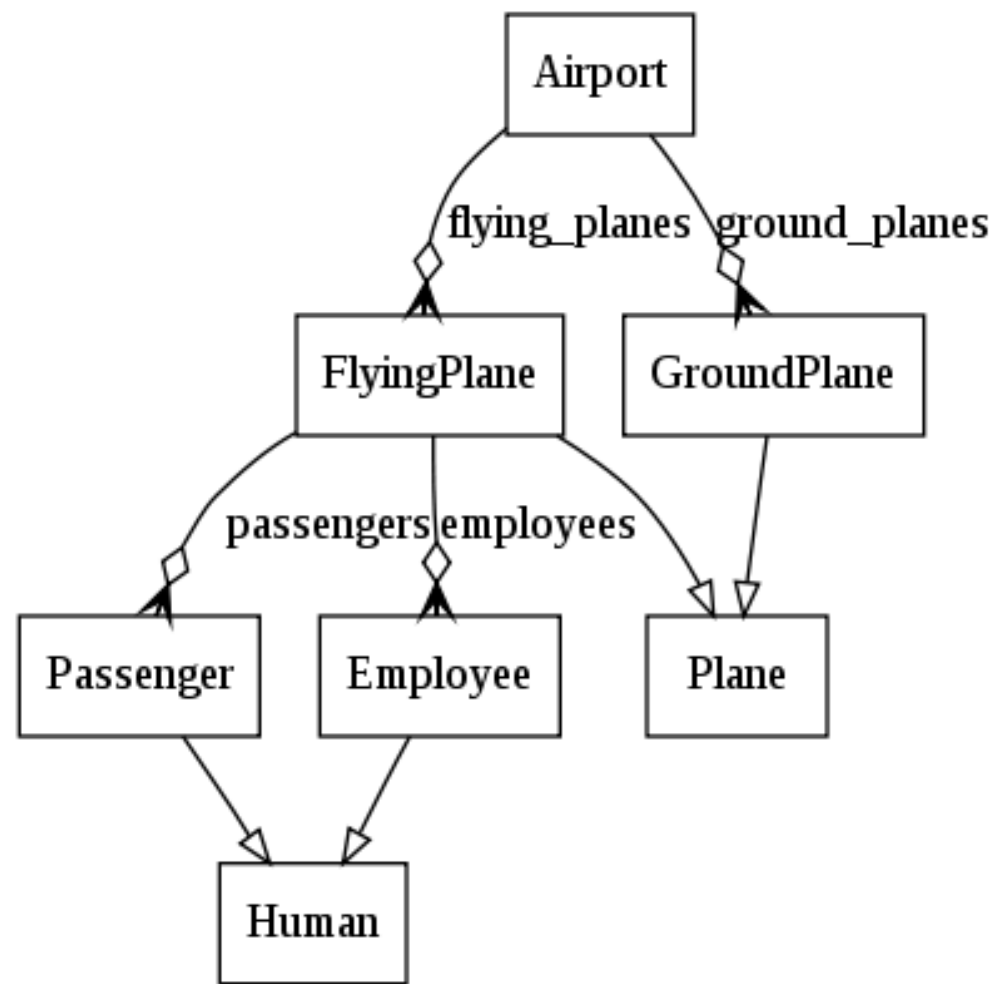
```
class A(object):
    def foo(self):
        print('A')
class B(A):
    def foo(self):
        print('B')
        super(B, self).foo()
class C(A):
    def foo(self):
        print('C')
        super(C, self).foo()
class D(B, C):
    def foo(self):
        print('D')
        super(D, self).foo()
d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

D
B
C
A

Class Diagrams

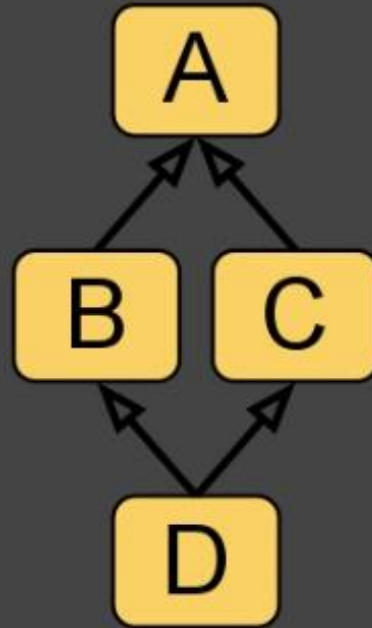
Class diagrams are visual representations of the relationships among classes

- 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



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
```



Super Function

SECTION 5

Python super()

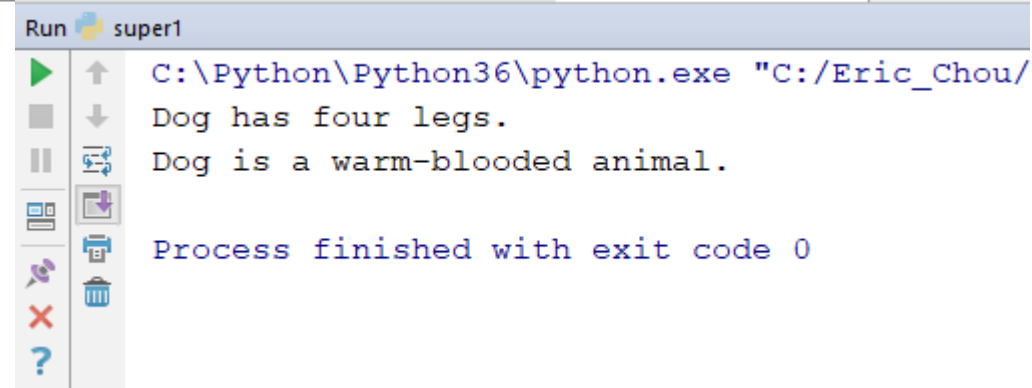
- The **super()** builtin returns a proxy object that allows you to refer parent class by 'super'.
- In Python, **super()** built-in has two major use cases:
 1. Allows us to avoid using base class explicitly
 2. Working with Multiple Inheritance

super() with Single Inheritance Demo

Program: super1.py

Go PyCharm!!!

```
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()
```



```
Run super1  
C:\Python\Python36\python.exe "C:/Eric_Chou/  
Dog has four legs.  
Dog is a warm-blooded animal.  
  
Process finished with exit code 0
```

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).

```
# 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!!!

```

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)
```

```

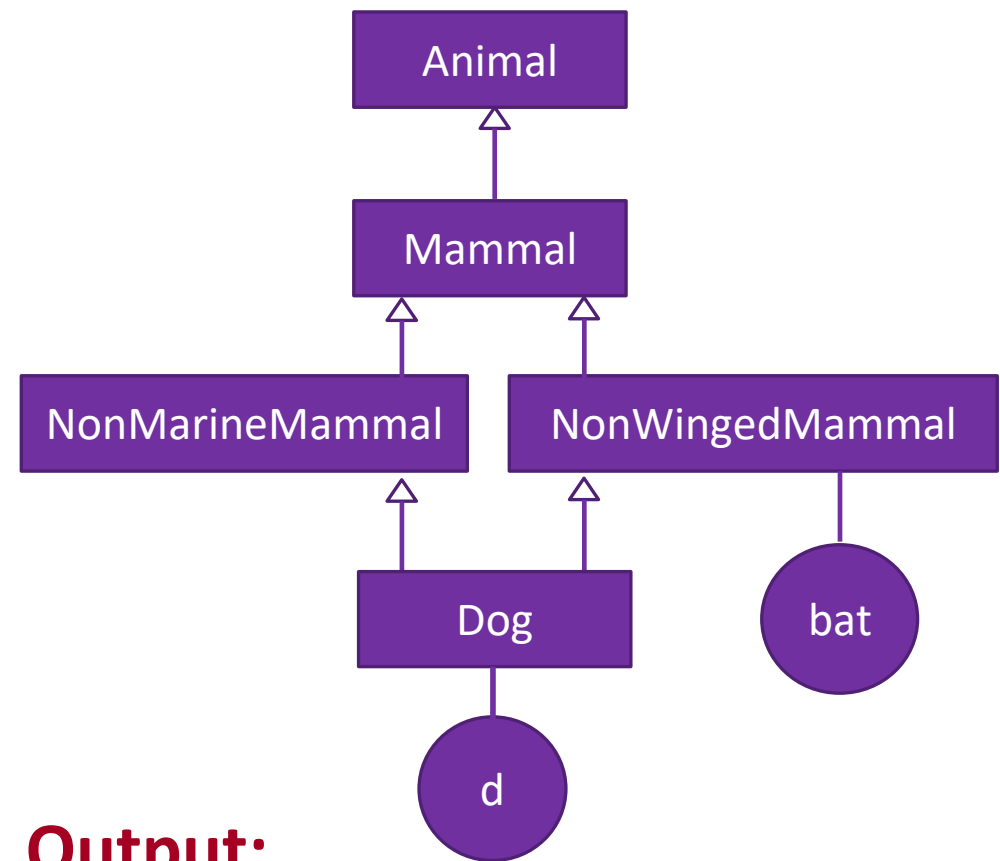
class Dog(NonMarineMammal, NonWingedMammal):
    def __init__(self):
        print('Dog has 4 legs.')
```

```

        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 class is always called before the method of the base class.

In our example, Dog class is called before **NonMarineMammal** or **NonWingedMammal**. 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

SECTION 6

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: cmdline1.py

Go PyCharm!!!

```
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 cmdline1.py A B C D E
Number of arguments: 6 arguments.
Argument List: ['cmdline1.py', 'A', 'B', 'C', 'D', 'E']
```

Demo Program: cmdline2.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 cmdline2.py A B C D E
A
B
C
D
E
```

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

```
while ((c = getopt (argc, argv, "sh:f:p:mb")) != -1)
switch (c){
  case 's':
    suppress = 1;
    break;
  case 'h':
    oph = atoi(optarg);
    if (oph>0) ht_SIZE = oph;
    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.)

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

SECTION 7

*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.

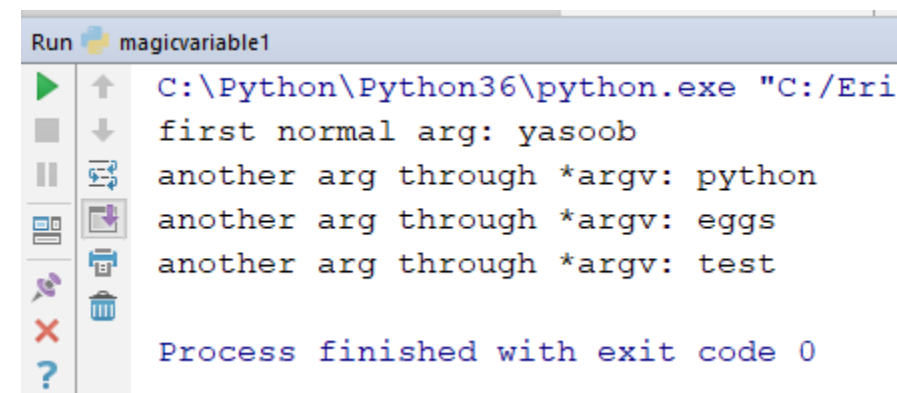
Usage of *args

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

```
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('yasooob', 'python', 'eggs', 'test')
```

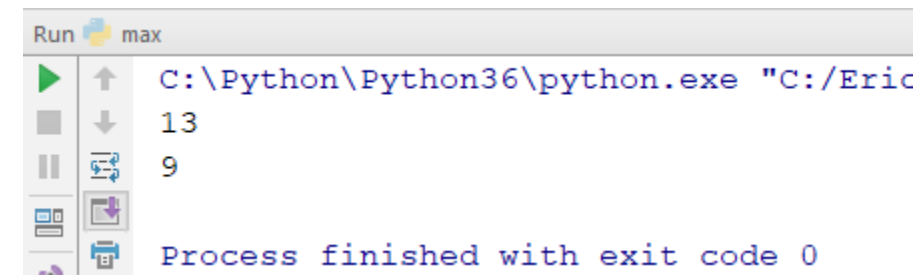


```
Run magicvariable1  
C:\Python\Python36\python.exe "C:/Eri  
first normal arg: yasooob  
another arg through *argv: python  
another arg through *argv: eggs  
another arg through *argv: test  
  
Process finished with exit code 0
```


Demo Program: max.py

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

```
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))
```



```
Run max  
C:\Python\Python36\python.exe "C:/Eric  
13  
9  
Process finished with exit code 0
```

Usage of **kwargs

variable number of keyworded arguments

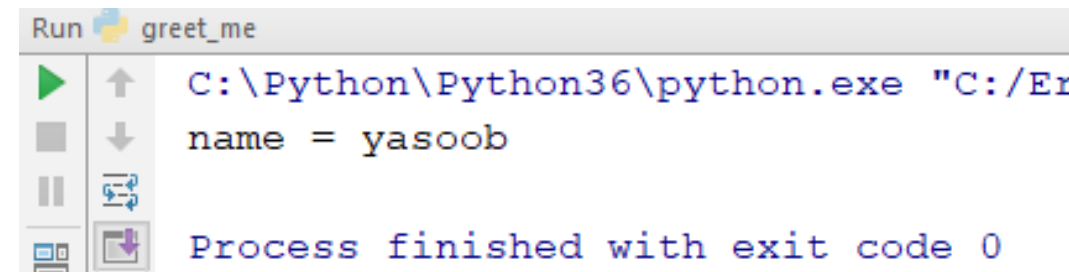
- **kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function. Here is an example to get you going with it:

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

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

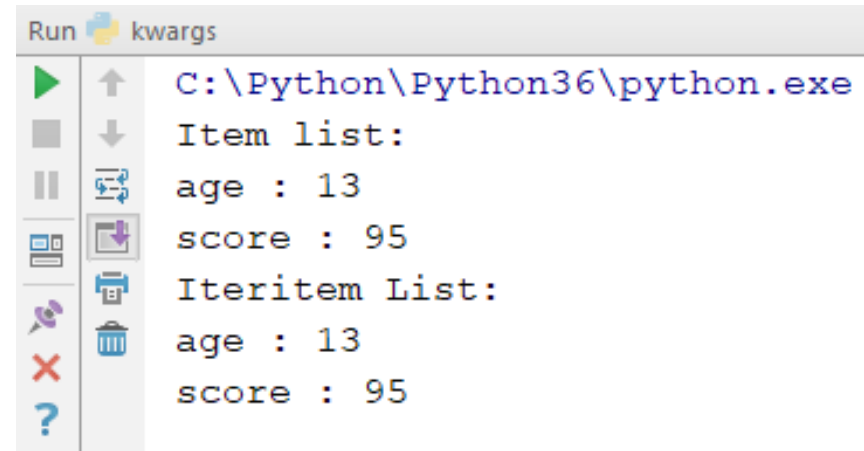
Demo Program: greet_me.py

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



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)
```



Run kwargs

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-dictionary **`f(**kwargs)`**

Augmented by tuple

`def f(*args)`

Augmented by dictionary

`def f(kwargs)`**

```

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

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

# augmented 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:

```
import someclass

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

someclass.get_info = get_info
```

Details

SECTION 8

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 ****kwargs**, 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, **kwargs):
        self.shapename = shapename
        super().__init__(**kwargs)

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

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


Abstract Method

SECTION 9

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:
- 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 ABC
class MyABC(ABC):
    pass
```

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

How ABCs Work?

Create Abstract Method (AbstractOperation.py)

```
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
```

```

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())

```

Demo Program: ConcreteOperations.py

```

Run ConcreteOperationClasses
C:\Python\Python36\python.exe "C:/Eric_Chau/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!!!

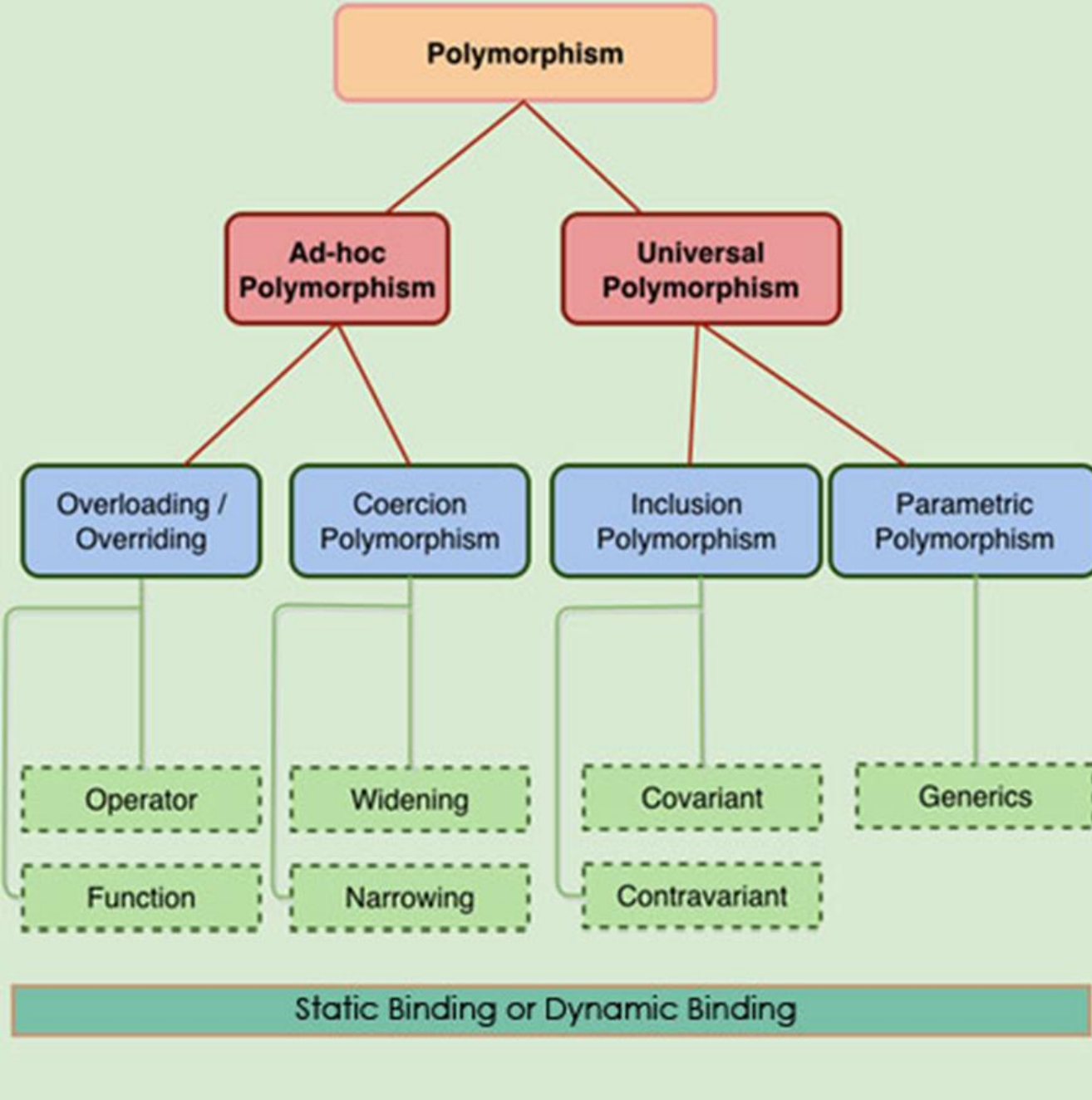
To be done.

Polymorphism

SECTION 10

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

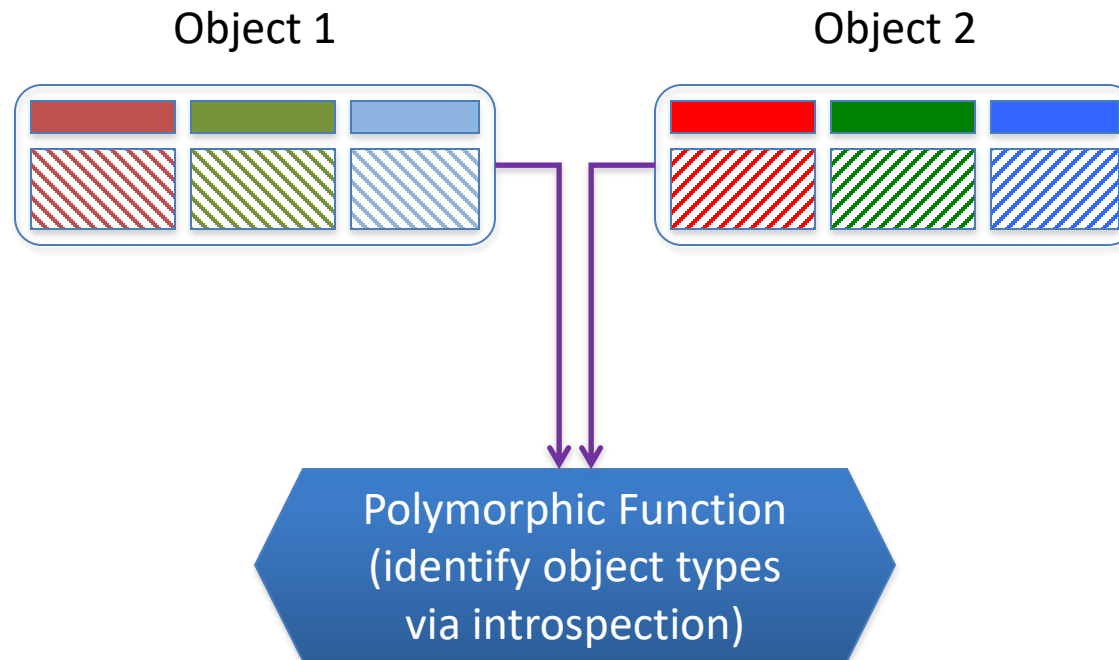
Python Polymorphism

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

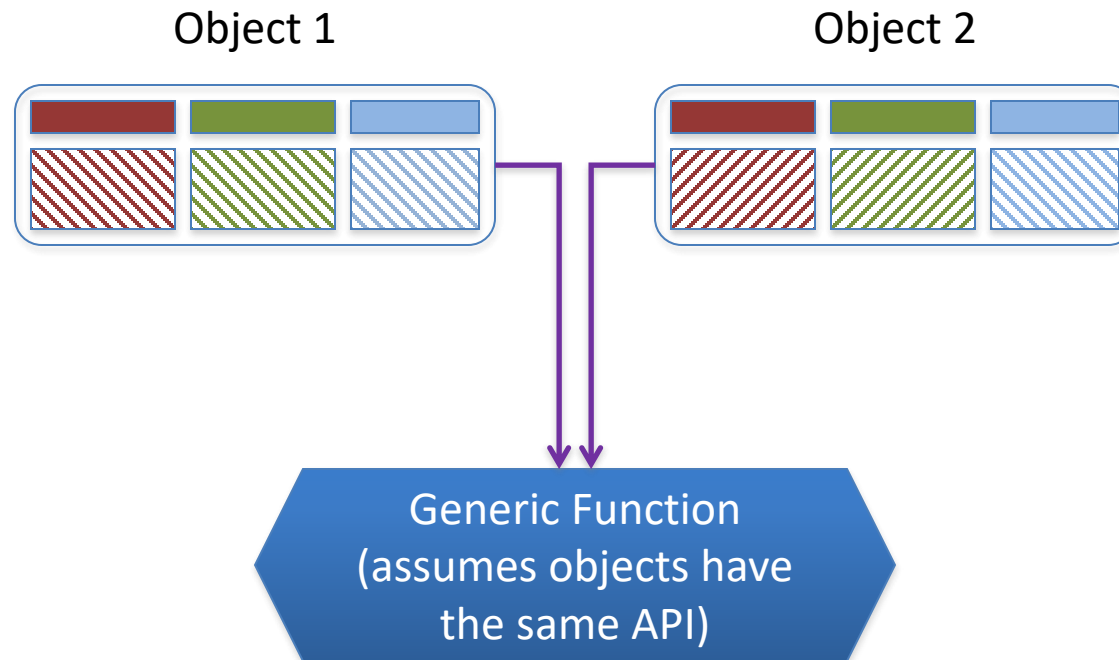
Polymorphism

- 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](#)

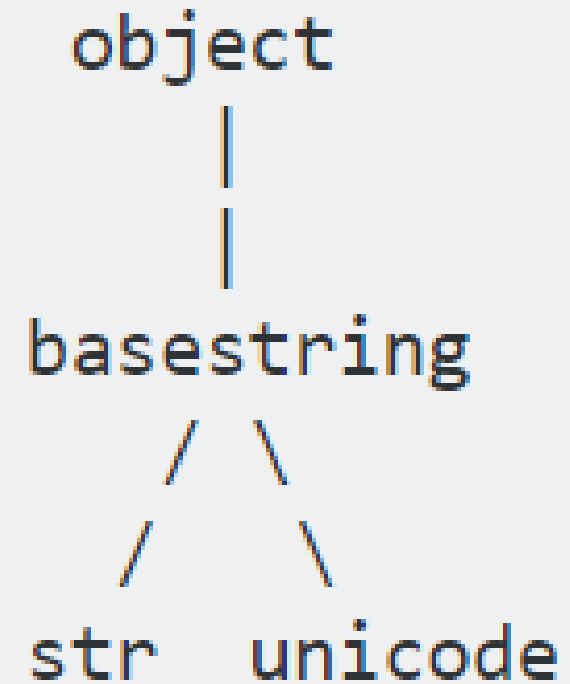
Python provides several ways of identifying data types:

- **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()))
```

Python 3: basestring is replaced by str

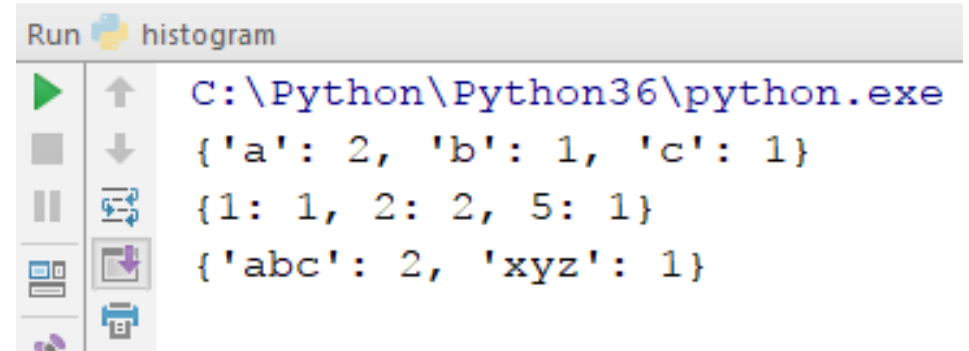


Polymorphic Syntax

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"))
```



The screenshot shows a Python IDE's Run console window titled "Run histogram". The console displays the output of the histogram function for three different inputs. The first line shows the output for the string "aabc", which is a dictionary with keys 'a', 'b', and 'c' and values 2, 1, and 1 respectively. The second line shows the output for the list [1, 2, 2, 5], which is a dictionary with keys 1, 2, and 5 and values 1, 2, and 1 respectively. The third line shows the output for the tuple ("abc", "abc", "xyz"), which is a dictionary with keys 'abc' and 'xyz' and values 2 and 1 respectively. The console window has a toolbar on the left with icons for running, stepping through, and other debugging actions.

```
Run histogram  
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 in 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

```
# 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
```

```
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: ")
```

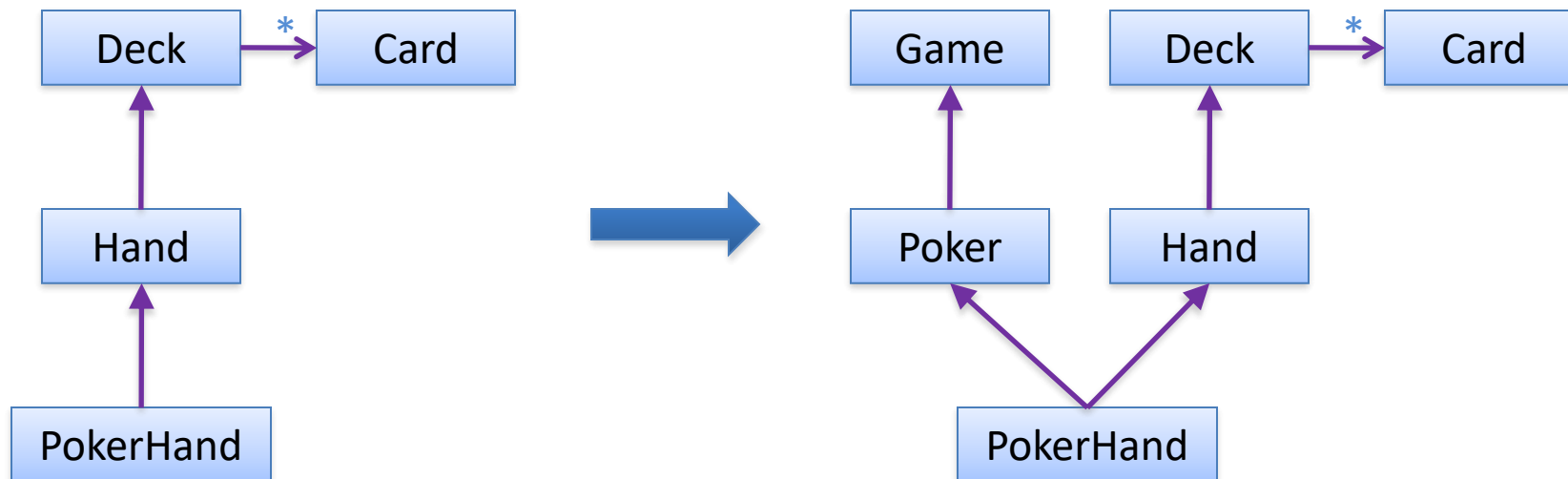
```
print_series(InfiniteSeries())
```

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

SECTION 11

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.

The sys module

sys.platform

sys.version

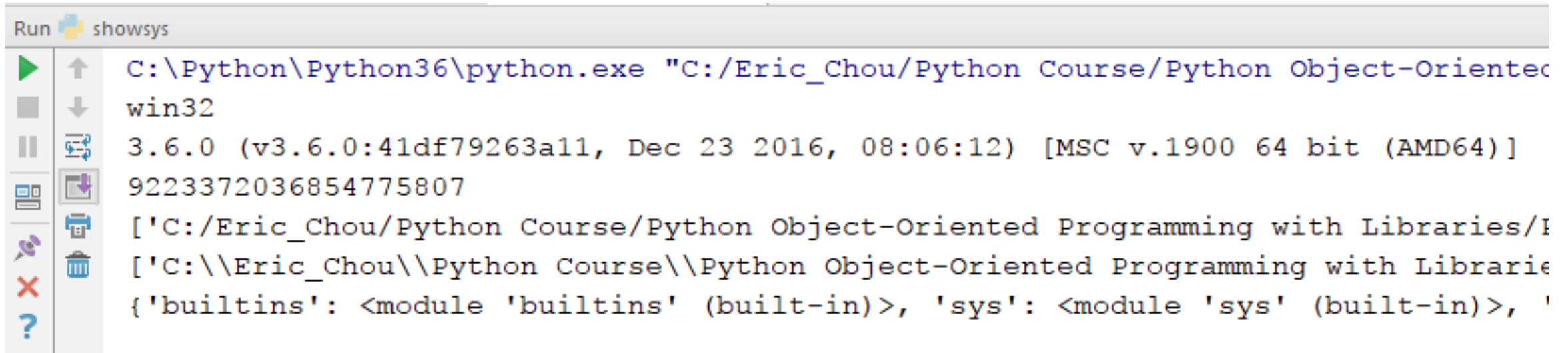
sys.maxsize
sys.maxint not
in Python3

sys.argv

sys.path

sys.modules

Go PyCharm!!!



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.

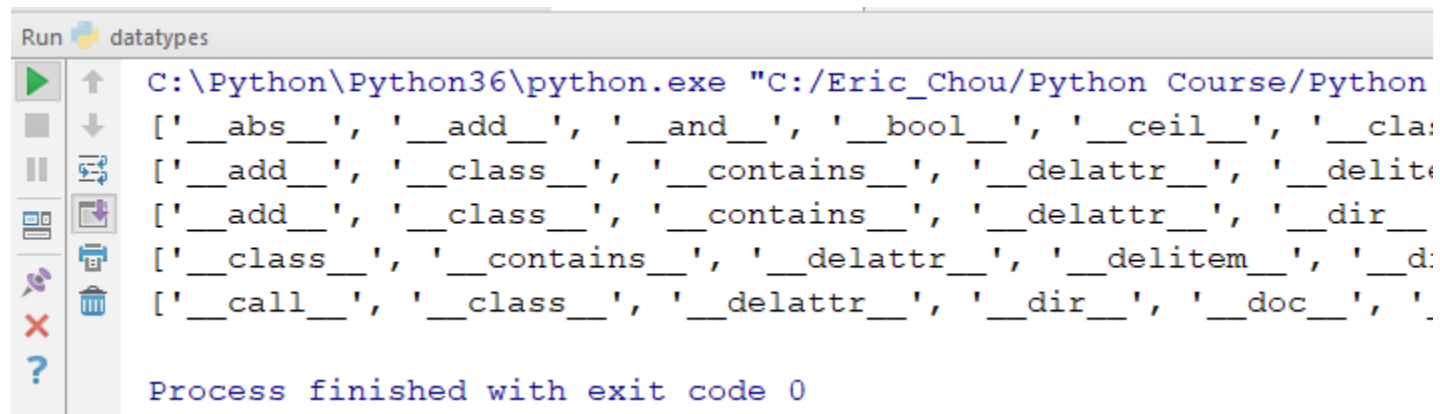
- **dir(keyword)**: The keyword module's attributes
- **dir(sys)**: The sys module's attributes
- **dir()**: Names in the current scope
- **__builtins__**: `__builtins__` appears to be a name in the current scope that's bound to the module object named `__builtin__`.
- **dir(__builtins__)**: The `__builtins__`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



The screenshot shows the PyCharm Run console for a program named 'datatypes'. The command executed is 'C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python...'. The output lists the attributes of various Python objects: integers (42), lists, tuples, dictionaries, and the 'dir' function itself. The console also shows the status 'Process finished with exit code 0'.

```
Run datatypes
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__clas
['__add__', '__class__', '__contains__', '__delattr__', '__delite
['__add__', '__class__', '__contains__', '__delattr__', '__dir__
['__class__', '__contains__', '__delattr__', '__delitem__', '__d
['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '
Process finished with exit code 0
```

```

1  >>> class Person(object):
2  ...     """Person class."""
3  ...     def __init__(self, name, age):
4  ...         self.name = name
5  ...         self.age = age
6  ...     def intro(self):
7  ...         """Return an introduction."""
8  ...         return "Hello, my name is %s and I'm %s." % (self.name, self.age)
9  ...
10 >>> bob = Person("Robert", 35)    # Create a Person instance
11 >>> joe = Person("Joseph", 17)    # Create another
12 >>> joe.sport = "football"        # Assign a new attribute to one instance
13 >>> dir(Person)                   # Attributes of the Person class
14 ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
15  '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
16  '__setattr__', '__str__', '__weakref__', 'intro']
17 >>> dir(bob)                      # Attributes of bob
18 ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
19  '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
20  '__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name']
21 >>> dir(joe)                      # Note that joe has an additional attribute
22 ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
23  '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
24  '__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name', 'sport']
25 >>> bob.intro()                   # Calling bob's intro method
26 "Hello, my name is Robert and I'm 35."
27 >>> dir(bob.intro)                # Attributes of the intro method
28 ['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__get__',
29  '__getattribute__', '__hash__', '__init__', '__new__', '__reduce__',
30  '__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

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

```
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())
```



```
{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
```

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>
__getattr__: <slot wrapper '__getattr__' 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>
```

Duck Typing (Interface)

SECTION 12

Python duck typing (or automatic interfaces)

- Duck typing is a feature of a type system where the semantics of a class is determined by his ability to respond to some message (method or property).
- The canonical example (and the reason behind the name) is the duck test: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
- Duck Typing needs to be implemented by
 - **Type Inference (Type Propagation)**










Polymorphic Method

A method of same name owned by different objects.

- **Goose()** and **Duck()** create two objects of different types. They shared a same polymorphic method.
- Python does not require type declaration. The object type is assigned at run-time (dynamic type assignment).

```
1  # duck0.py
2  # This example shows a polymorphic method
3  class Duck:
4      def quack(self):
5          print('Quack!')
6  class Goose:
7      def quack(self):
8          print('Quack')
9
10  Goose().quack()    # Goose() create instant temporary object
11  Duck().quack()     # Duck() create instant temporary object
```

Run  duck0		
		C:\Pytl
		Quack
		Quack!

Interface (Automatic Interfaces)

A interface is a reference which can represent objects of different types.

Generic Methods

Methods which can be operated on objects of different type. (Or, objects of same interface.)

Java Interface:

Reference of Objects of different classes sharing the same polymorphic methods.

```
interface IEngine {  
    void turnOn();  
}  
  
public class EngineV1 implements IEngine {  
    public void turnOn() {  
        // do something here  
    }  
}  
  
public class Car {  
    public Car(IEngine engine) {  
        this.engine = engine;  
    }  
  
    public void run() {  
        this.engine.turnOn();  
    }  
}
```

Python Class:

No need to declare interface. Just define polymorphic method. (Methods of same name)

This is called **Duck Typing**.

Duck Typing: methods of same name and working similarly. (Walk like ducks. Swim like ducks.)

```
class Car:
    def __init__(self, engine):
        self.engine = engine

    def run():
        self.engine.turn_on()
```

Duck Typing Example

```
class Duck:
    def quack(self):          ← Polymorphic Method
        print "Quack, quack!"
    def fly(self):           ← Polymorphic Method
        print "Flap, Flap!"

class Person:
    def quack(self):
        print "I'm Quackin'!"
    def fly(self):
        print "I'm Flyin'!"

def in_the_forest(thing):    ← Generic Method
    thing.quack()
    thing.fly()

in_the_forest(Duck())
in_the_forest(Person())
```

Demo Program: duck1.py

Go PyCharm!!!

```

1  class Duck:
2      def quack(self):
3          print("Quack, quack!")
4      def fly(self):
5          print("Flap, Flap!")
6
7  class Goose:
8      def quack(self):
9          print("Quack!")
10     def fly(self):
11         print("Flap!")
12
13 class Person:
14     def quack(self):
15         print("I'm Quacking!")
16     def fly(self):
17         print("I'm Flying!")
18
19 # Generic Method: A method handle different object types
20
21 def in_the_forest(thing):
22     thing.quack()      # both quack() and fly() are polymorphic methods
23
24     thing.fly()
25
26 in_the_forest(Duck())
27 in_the_forest(Goose())
28 in_the_forest(Person())

```

Run  duck1



C:\Python\Python

Quack, quack!

Flap, Flap!

Quack!

Flap!

I'm Quacking!

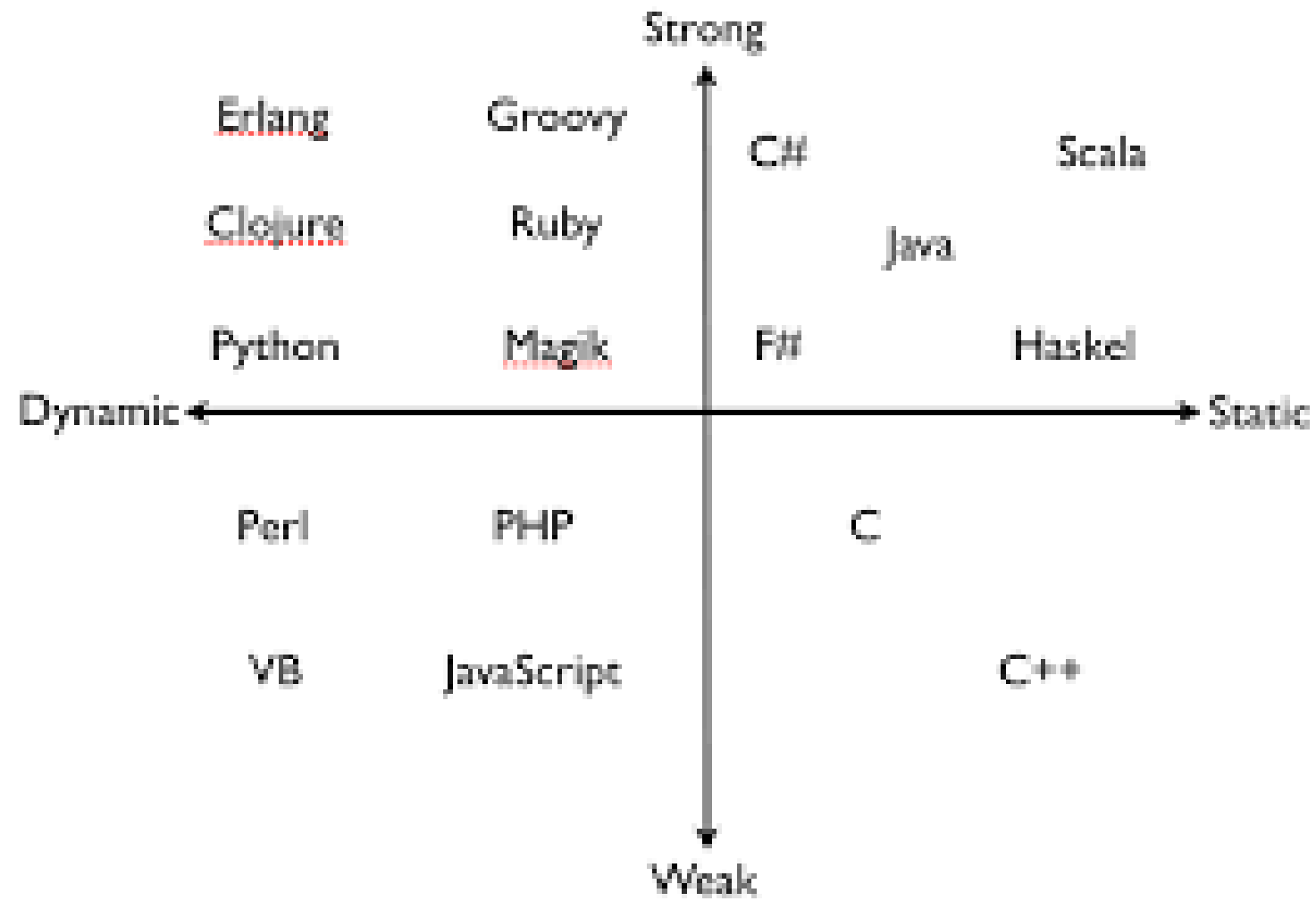
I'm Flying!

Python Types

SECTION 13

Python Dynamic Typing

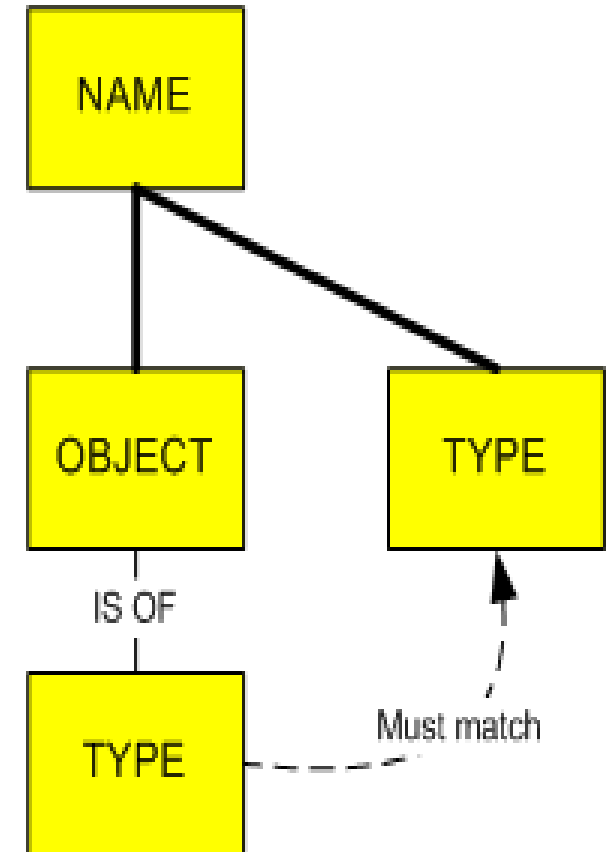
- Python is a dynamically-typed language.
- Java is a statically-typed language.
- Python is a weak-typed (weaker) language.
- Java is a strong-typed (stronger) language.
- Python has type inference. Java does not.



In a **statically typed language**, every variable name is bound both

- to a type (at compile time, by means of a data declaration)
- to an object.

The binding to an object is optional — if a name is not bound to an object, the name is said to be *null*.

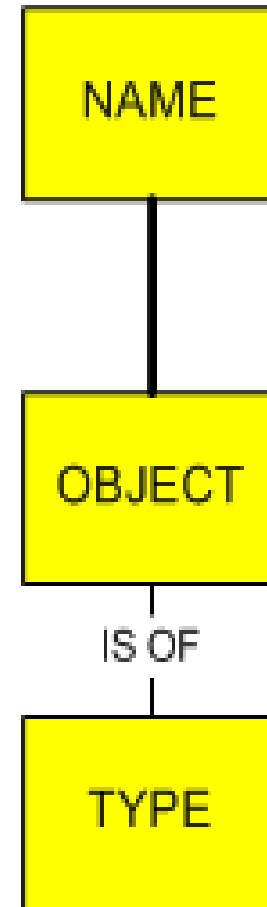


Note:

Once a variable name has been bound to a type (that is, declared) it can be bound (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception.

In a **dynamically typed language**, every variable name is (unless it is null) bound only to an object.

Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program.



Python Variables can be assigned with Values of Different Types

- We've noticed before that Python is happy to allow us to store any kind of value in any variable we'd like.

```
x = 3
```

```
y = 'Boo'
```

```
z = [1, 2, 3]
```

- We've also seen that we can potentially change the type of a variable any time we'd like by simply assigning a value of a different type into it.

```
x = (1, 2)    # x is now a tuple
```

```
y = 9.5      # y is now a float
```

```
z = 'Alex'   # z is now a str
```

Python Types is Associated with Right-Hand-Side Value. Not Left-Hand-Side Reference.

- Or, thought differently, variables themselves don't have types at all in Python; only the values of those variables have types.
- If we use a variable after assigning it a value, what we're allowed to do with it — the operators we can use, the **functions** into which we can pass it as an argument, and so on — is determined by the type of its value at the time we use it.
- Forming interface automatically with polymorphic methods. (Duck Typing) Otherwise, exceptions will be raised.

With Polymorphic Methods

```
w = 'Alex'
```

```
print(len(w)) # prints 4
```

```
q = 57
```

```
print(len(q)) # raises an exception, because ints don't have a length
```

Use of Duck Typing

SECTION 14

Use of Duck Typing

1. **Declare a global variable** and assign value of different type later.
2. **Polymorphic method:** object behaves similarly.
3. **Generic method** (Forming interface automatically):
Generic method can only operate on objects which has polymorphic methods (methods of same name).
4. **Generic Container:** generic data structures (list, tree, graph, matrix, hash table, set, map, associated memory)

How duck typing affects the way we write functions

```
def foo(x, y):  
    return x.bar(y) * 2
```

Type Inference

- Let's left the types out of the function's signature, because it's not as clear what they are until we stop to think about it.
- What must be true about the types of x and y in order to successfully evaluate a call to foo(x, y)?

How duck typing affects the way we write functions

- **x** must be an **object** of some class that has a method called **bar** that takes one parameter (in addition to **self**). There might be many classes like this, and it may not always be the case that all their **bar** methods even do the same thing; the presence of the method is one part of what makes this legal.
- **y** must have a type that is compatible as an **argument** to **bar**. Depending on **x**'s type — and depending on what its **bar** method does — this constraint will be different. Any combination that works is potentially legal.

How duck typing affects the way we write functions

- The type of value returned from the **bar** method must be something that can be multiplied by 2.
- At first blush, that sounds like it must be a number, but if you think harder, you'll remember that you can also multiply other kinds of things (e.g., lists, strings) by numbers, too. (If that sounds weird to you, try evaluating `[1, 2, 3] * 2` in a Python interpreter and see what you get back.)
- See Type Propagation in the next section:
 1. **Object type.**
 2. **Argument**
 3. **Operand type**

Type Inference

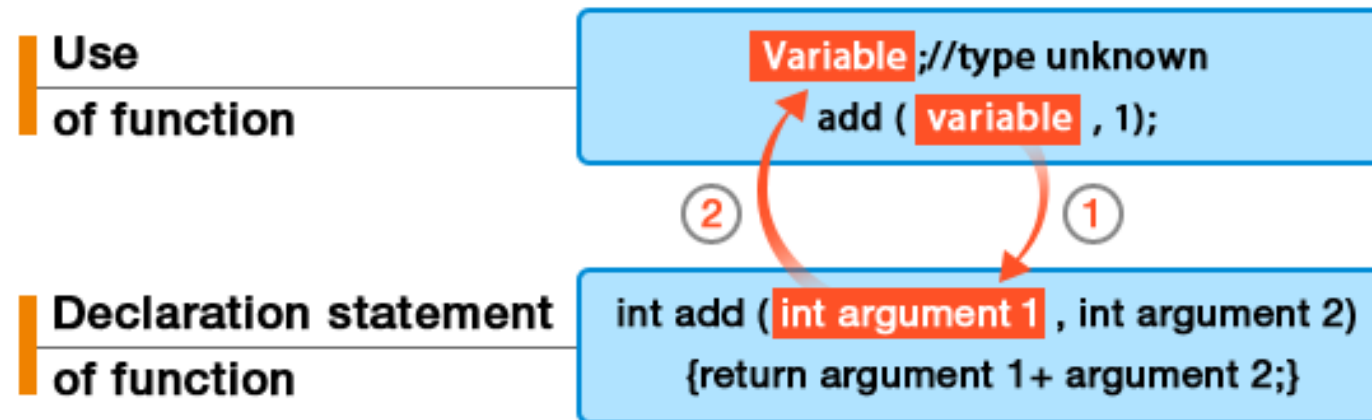
SECTION 15

Type Inference

- What determine the type of overall expression? (Type Checking for the Output)
- The result of an arithmetic operator usually has the same type as the operands. The result of a comparison is usually Boolean.
- The result of a function call has the type declared in the function's header. The result of an assignment (in languages in which assignments are expressions) has the same type as the left-hand side. In a few cases, however, the answer is not obvious. In particular, operations on subranges and on composite objects do not necessarily preserve the types of the operands.

Type Inference Permeates Modern Languages

The type of variable is inferred to be int from the declaration statement



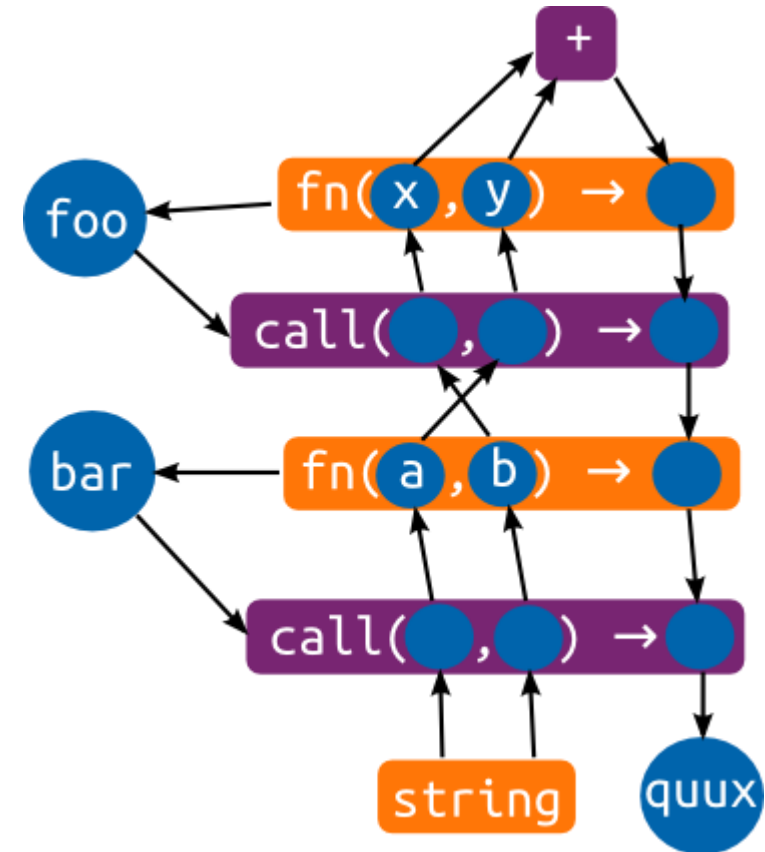
Type inference is a function which automatically specifies the type that can perform “inference” if the necessary minimum type is specified (Figure). If this function is used, the advantage where errors can be checked at an early stage can be realized, without specifying the type for all of the variables and functions. [Java 8, Swift, Haskell, Scala]

Type Propagation

```
function foo(x, y) { return (x + y); }  
function bar(a, b) { return foo(b, a); }  
var quux = bar("goodbye", "hello");
```

Type Propagation:

You can see the function types, as orange boxes, containing (references to) abstract values. Function declarations will cause such types to be created, and added to the variable that names the function. The purple boxes are propagation strategies. There are two calls in the program, corresponding to the two purple call boxes. At the top is a simple box that handles the + operator. If a string is propagated to it, it'll output a string type, and if two number types are received, it'll output a number type.



Typing

SECTION 16

Polymorphic Method Design

Can we write a makelist function to replace
`list(collection_data_type)`

Key Point:

- Use built-in functions, language structures, and operators to achieve the polymorphism you want to achieve

list(collection_type)

Generic Method

```
>>> list([1, 2, 3])      # you can pass it a list
```

```
[1, 2, 3]
```

```
>>> list((1, 2, 3))      # you can also pass it a tuple
```

```
[1, 2, 3]
```

```
>>> list({'a', 'b', 'c'}) # or even a set
```

```
['b', 'a', 'c']        # (remember that sets are not ordered)
```

Polymorphic Language Structure (for-loop)

```
for x in [1, 2, 3]:  
    print(x)
```

```
1  
2  
3
```

```
for x in (1, 2, 3):  
    print(x)
```

```
1  
2  
3
```

```
# range() return tuple  
for x in range(5):  
    print(x)
```

```
1  
2  
3  
4  
5
```

```
for x in 3:  
    print(x)
```

```
Traceback (most recent call last):
```

```
File "", line 1, in
```

```
    for x in 3:
```

```
TypeError: 'int' object is not iterable
```

Implementation of makelist

```
def makelist(items):  
    the_list = []  
  
    for x in items:  
        the_list.append(x)  
  
    return the_list
```

And if we try this function out, we'll see it hits the nail right on the head:

```
>>> makelist([1, 2, 3])  
[1, 2, 3]  
>>> makelist((1, 2, 3))  
[1, 2, 3]  
>>> makelist({'a', 'b', 'c'})  
['b', 'a', 'c']  
>>> makelist(range(5))  
[0, 1, 2, 3, 4]
```

Function Closure

SECTION 17

Function Closure

A **function closure** is an object (interface) that have the same **polymorphic** method so that, the object can be passed as a parameter to a **generic method** (which operator on different object types with same polymorphic method).

The function closures can be run by the same generic function. (Same as the Java Calculus package developed by Dr. Eric Chou).

ducktyping2.py

```
1 import math
2 class ZeroCalc:
3     def calculate(self, n):
4         return 0
5 class SquareCalc:
6     def calculate(self, n):
7         return n * n
8 class CubeCalc:
9     def calculate(self, n):
10        return n * n * n
11 class LengthCalc:
12     def calculate(self, n):
13         return len(n)
14 class SquareRootCalc:
15     def calculate(self, n):
16         return math.sqrt(n)
17 class MultiplyByCalc:
18     def __init__(self, multiplier):
19         self._multiplier = multiplier
20     def calculate(self, n):
21         return n * self._multiplier
22
23 def run_calcs(calcs: ['Calc'], starting_value):
24     current_value = starting_value
25     for calc in calcs:
26         current_value = calc.calculate(current_value)
27     return current_value
28
29 a1 = run_calcs([SquareCalc(), SquareCalc()], 4)
30 print(a1)
31 a2 = run_calcs([LengthCalc(), MultiplyByCalc(2)], 'Boo')
32 print(a2)
33 a3 = run_calcs([], 80)
34 print(a3)
35 a4 = run_calcs([MultiplyByCalc(3), LengthCalc(), SquareCalc()], 'Boo')
36 print(a4)
37
```

list of function closure.

A **function closure** is an **object** which has a polymorphic calculate method.

```
Run ducktyping2
C:\Python\Python36\python.exe
256
6
80
81
```

Iterables vs. Iterators vs. Generators

SECTION 18

Iterables vs. Iterators vs. Generators

A little pocket reference on iterables, iterators and generators.

The following related concepts in Python are very confusing:

- a container
- an iterable
- an iterator
- a generator
- a generator expression
- a {list, set, dict} comprehension

a generator
expression



is

a generator



always is

an iterator



next()

*lazily produce
next value*

is

a generator
function



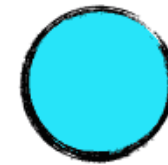
always is

iter()



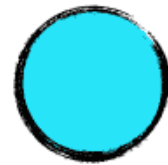
(an) iterable

typically is



a container

produces



{list, set, dict}
comprehension

Iterables

SECTION 19

Iterables

- As said, most containers are also iterable. But many more things are iterable as well. Examples are open files, open sockets, etc. Where containers are typically finite, an iterable may just as well represent an infinite source of data.
- An iterable is any object, not necessarily a data structure, that can return an iterator (with the purpose of returning all of its elements). That sounds a bit awkward, but there is an important difference between an iterable and an iterator. Take a look at this example:

```
>>> x = [1, 2, 3]
>>> y = iter(x)
>>> z = iter(x)
>>> next(y)
1
>>> next(y)
2
>>> next(z)
1
>>> type(x)
<class 'list'>
>>> type(y)
<class 'list_iterator'>
```

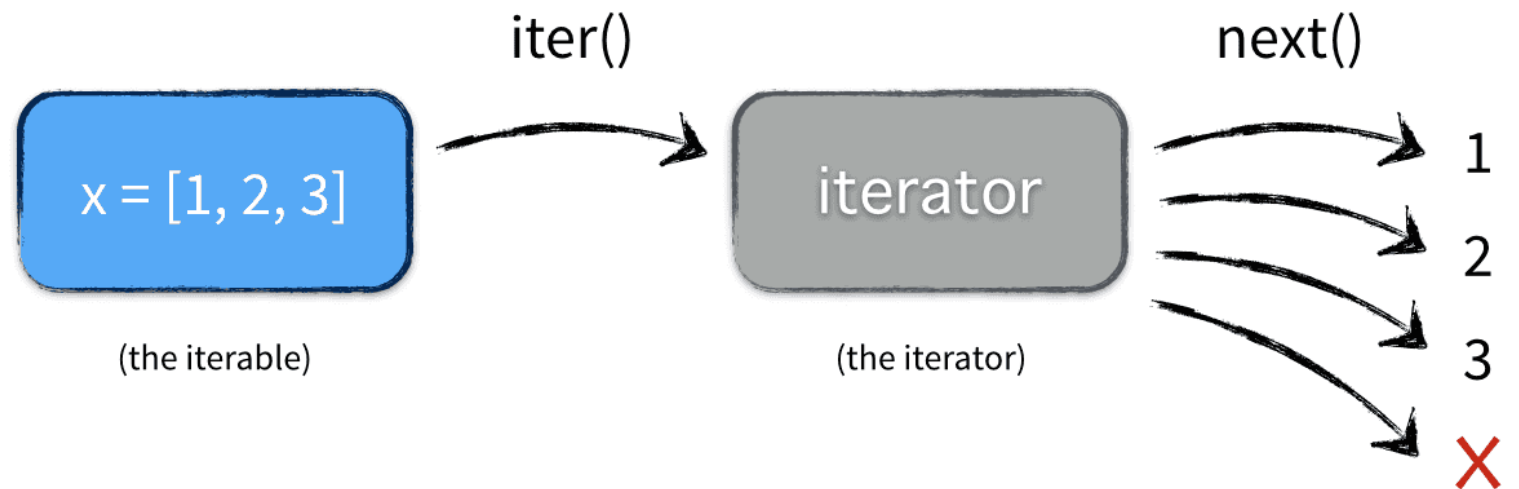
Here, x is the iterable, while y and z are two individual instances of an iterator, producing values from the iterable x. Both y and z hold state, as you can see from the example. In this example, x is a data structure (a list), but that is not a requirement.

NOTE:

Often, for pragmatic reasons, iterable classes will implement both `__iter__()` and `__next__()` in the same class, and have `__iter__()` return self, which makes the class both an iterable and its own iterator. It is perfectly fine to return a different object as the iterator, though.

For-Each Loop

```
x = [1, 2, 3]  
for elem in x:  
    ...
```



Disassemble Python Code for Iterator

- When you disassemble this Python code, you can see the explicit call to `GET_ITER`, which is essentially like invoking `iter(x)`. The `FOR_ITER` is an instruction that will do the equivalent of calling `next()` repeatedly to get every element, but this does not show from the byte code instructions because it's optimized for speed in the interpreter.

Disassemble Python Code for Iterator

disassemble.py

```
>>> import dis
>>> x = [1, 2, 3]
>>> dis.dis('for _ in x: pass')
      1      0 SETUP_LOOP                    14 (to 17)
          3 LOAD_NAME                      0 (x)
          6 GET_ITER
      >>    7 FOR_ITER                        6 (to 16)
          10 STORE_NAME                     1 (_)
          13 JUMP_ABSOLUTE                  7
      >>   16 POP_BLOCK
      >>   17 LOAD_CONST                      0 (None)
          20 RETURN_VALUE
```


Iterator

SECTION 20

Iterators

- So, what is an iterator then? It's a stateful helper object that will produce the next value when you call **next()** on it. Any object that has a **__next__()** method is therefore an iterator. How it produces a value is irrelevant.
- So, an iterator is a value factory. Each time you ask it for "the next" value, it knows how to compute it because it holds internal state.

Iterators

- There are countless examples of iterators. All of the itertools functions return iterators. Some produce infinite sequences:

```
>>> from itertools import count
>>> counter = count(start=13)
>>> (counter)
13
>>> (counter)
14
```

Iterators

```
from itertools import count
counter = count(start=13)
print((counter))
print(next(counter))
print(next(counter))
```

iterator2.py

```
count(13)
13
14
```

Cyclic Iterator

- Some produce infinite sequences from finite sequences:

```
from itertools import cycle
colors = cycle(['red', 'white', 'blue'])
print(next(colors))
print(next(colors))
print(next(colors))
print(next(colors))
```

iterator3.py

```
red
white
blue
red
```

Cyclic Iterator

- Some produce finite sequences from infinite sequences:

```
from itertools import islice, cycle
colors = cycle(['red', 'white', 'blue']) # infinite
limited = islice(colors, 0, 7)           # finite
for x in limited:                       # so safe to use for-loop on
    print(x)
```

iterator4.py

red
white
blue
red
white
blue
red

Example for Iterable and Iterator

```
from itertools import islice
class fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1
    def __iter__(self):
        return self
    def __next__(self):
        value = self.curr
        self.curr += self.prev
        self.prev = value
        return value

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

iterator5.py

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Example for Iterable and Iterator

- Note that this class is both an iterable (because it sports an `__iter__()` method), and its own iterator (because it has a `__next__()` method).
- The state inside this iterator is fully kept inside the `prev` and `curr` instance variables, and are used for subsequent calls to the iterator. Every call to `next()` does two important things:
 1. Modify its state for the next `next()` call;
 2. Produce the result for the current call.

Generators

SECTION 21

Generators

- Finally, we've arrived at our destination! The generators are my absolute favorite Python language feature. **A generator is a special kind of iterator**—the elegant kind.
- A generator allows you to write iterators much like the Fibonacci sequence iterator example above, but in an elegant succinct syntax that avoids writing classes with `__iter__()` and `__next__()` methods.
- Let's be explicit:
 - Any generator also is an iterator (not vice versa!);
 - Any generator, therefore, is a factory that lazily produces values.

Central idea: a lazy factory

From the outside, the iterator is like a lazy factory that is idle until you ask it for a value, which is when it starts to buzz and produce a single value, after which it turns idle again.

Generating Function

```
generator1.py
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

from itertools import islice
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

Generating Function

- Wow, isn't that elegant? Notice the magic keyword that's responsible for the beauty:

yield

- Let's break down what happened here: first of all, take note that fib is defined as a normal Python function, nothing special. Notice, however, that there's no return keyword inside the function body. The return value of the function will be a generator (read: an iterator, a factory, a stateful helper object).

Generating Function

- Now when `f=fib()` is called, the generator (the factory) is instantiated and returned. No code will be executed at this point: the generator starts in an idle state initially. To be explicit: the line `prev, curr = 0, 1` is not executed yet.
- Then, this generator instance is wrapped in an `islice()`. This is itself also an iterator, so idle initially. Nothing happens, still.
- Then, this iterator is wrapped in a `list()`, which will consume all of its arguments and build a list from it. To do so, it will start calling `next()` on the `islice()` instance, which in turn will start calling `next()` on our `f` instance.

Generating Function

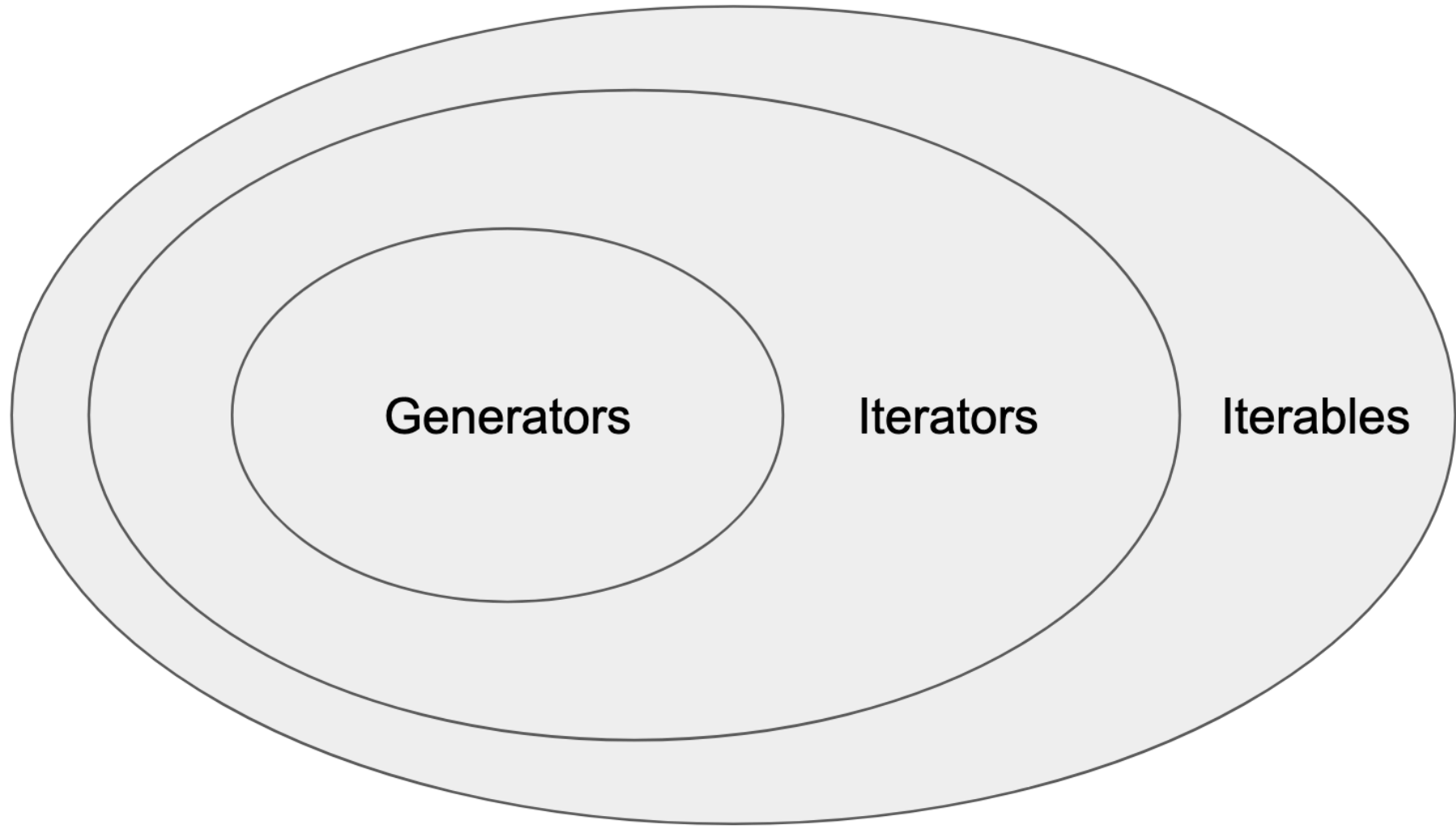
- But one step at a time. On the first invocation, the code will finally run a bit: `prev, curr = 0, 1` gets executed, the while True loop is entered, and then it encounters the `yield curr` statement. It will produce the value that's currently in the `curr` variable and become idle again.
- This value is passed to the `islice()` wrapper, which will produce it (because it's not past the 10th value yet), and list can add the value 1 to the list now.

Generating Function

- Then, it asks `islice()` for the next value, which will ask `f` for the next value, which will "unpause" `f` from its previous state, resuming with the statement `prev, curr = curr, prev + curr`. Then it re-enters the next iteration of the while loop, and hits the `yield curr` statement, returning the next value of `curr`.
- This happens until the output list is 10 elements long and when `list()` asks `islice()` for the 11th value, `islice()` will raise a **StopIteration** exception, indicating that the end has been reached, and `list` will return the result: a list of 10 items, containing the first 10 Fibonacci numbers. Notice that the generator doesn't receive the 11th `next()` call. In fact, it will not be used again, and will be garbage collected later.

Types of Generators

SECTION 22



Types of Generators

- There are two types of generators in Python: generator functions and generator expressions. A generator function is any function in which the keyword `yield` appears in its body. We just saw an example of that. The appearance of the keyword `yield` is enough to make the function a generator function.
- The other type of generators are the generator equivalent of a list comprehension. Its syntax is really elegant for a limited use case.



Python Generators

A Quick Guide for Beginners

```
def gen_func():
```

```
...
```

```
while <cond>:
```

```
...
```

```
yield num
```

```
next(gen_func())
```

```
gen_expr = (a**(1/2)
```

```
for a in alist)
```

```
pipeline
```

```
gen_fn1()
```

```
=> gen_fn2()
```

```
=> gen_fn3()
```

```
for item in  
gen_func(args):  
    print(item)
```



www.techbeamers.com

Comprehension Generators

generator2.py

```
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
b = [x*x for x in numbers]      # comprehensive list
c = {x*x for x in numbers}      # comprehensive set
d = {x:x*x for x in numbers}    # comprehensive dict
print(a)
print(b)
print(c)
print(d)

<generator object <genexpr> at 0x000001BE7126F048>
[1, 4, 9, 16, 25, 36]
{1, 4, 36, 9, 16, 25}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Generator expressions

- **A comprehension-based expression that results in an iterator object**
 - Does not result in a container of values
 - Must be surrounded by parentheses unless it is the sole argument of a function
 - May be returned as the result of a function

```
numbers = (random() for _ in range(42))  
sum(numbers)
```

```
sum(random() for _ in range(42))
```

Generator Expression

note: this is not a tuple comprehension

generator3.py

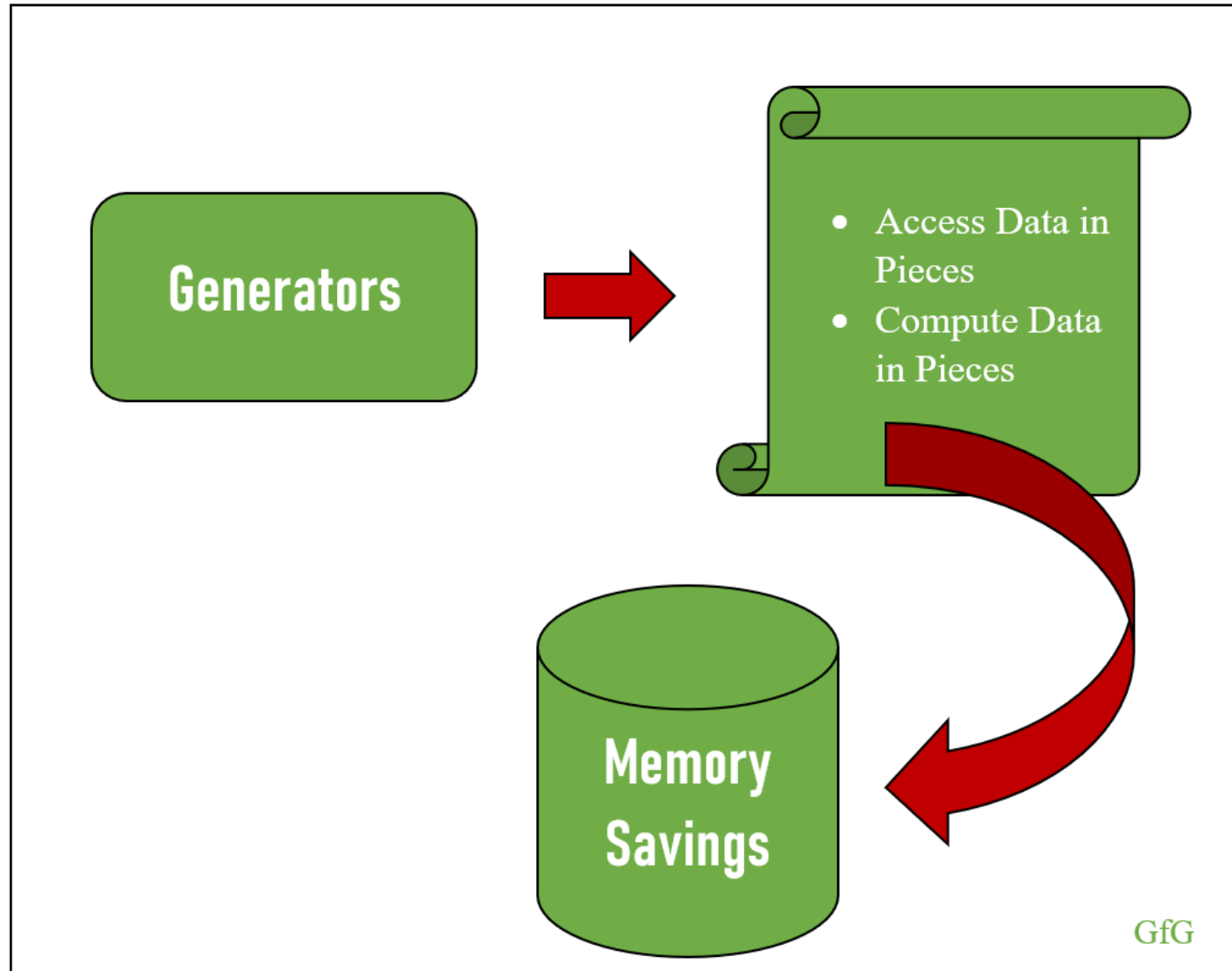
```
# Generator Expression
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

1
4
9
16

Generator Expression

note: this is not a tuple comprehension

- Note that, because we read the first value from `lazy_squares` with `next()`, its state is now at the "second" item, so when we consume it entirely by calling `list()`, that will only return the partial list of squares. (This is just to show the lazy behaviour.) This is as much a generator (and thus, an iterator) as the other examples above.



Summary for Generators

- Generators are an incredible powerful programming construct. They allow you to write streaming code with fewer intermediate variables and data structures. Besides that, they are more memory and CPU efficient. Finally, they tend to require fewer lines of code, too.

Generating a list

- Tip to get started with generators: find places in your code where you do the following:

```
def something():  
    result = []  
    for ... in ...:  
        result.append(x)  
    return result
```

Generating a list

- Tip to get started with generators: find places in your code where you do the following:

```
def iter_something():  
    for ... in ...:  
        yield x  
  
# def something():  
# Only if you really need a list structure  
# return list(iter_something())
```

Summary

SECTION 23

Summary

- This chapter goes over class-to-class relationship
- Its main purpose is to build a large software library and system
- Object-Oriented Design includes the following 3 major topics:
 - Class Design
 - Object-Oriented Thinking
 - Class to Class relationship



End of Chapter 5B
