GACRC

Python Language Basics II

Georgia Advanced Computing Resource Center
University of Georgia
Zhuofei Hou, HPC Trainer
zhuofei@uga.edu

GACRC

Outline

- What is GACRC?
- Program Structure: Control Flow and Loop
- Function: Procedural and Functional Programming
- Class: Object-Oriented Programming (OOP)

What is GACRC?

Who Are We?

- **G**eorgia **A**dvanced **C**omputing **R**esource **C**enter
- Collaboration between the Office of Vice President for Research (**OVPR**) and the Office of the Vice President for Information Technology (**OVPIT**)
- Guided by a faculty advisory committee (GACRC-AC)

Why Are We Here?

- To provide computing hardware and network infrastructure in support of high-performance computing (**HPC**) at UGA

Where Are We?

- http://gacrc.uga.edu (Web)
- http://wiki.gacrc.uga.edu (Wiki)
- http://gacrc.uga.edu/help/ (Web Help)
- https://wiki.gacrc.uga.edu/wiki/Getting_Help (Wiki Help)

GACRC Users September 2015

Colleges & Schools	Depts	PIs	Users
Franklin College of Arts and Sciences	14	117	661
College of Agricultural & Environmental Sciences	9	29	128
College of Engineering	1	12	33
School of Forestry & Natural Resources	1	12	31
College of Veterinary Medicine	4	12	29
College of Public Health	2	8	28
College of Education	2	5	20
Terry College of Business	3	5	10
School of Ecology	1	8	22
School of Public and International Affairs	1	3	3
College of Pharmacy	2	3	5
	40	214	970
Centers & Institutes	9	19	59
TOTALS:	49	233	1029

GACRC Users September 2015

Centers & Institutes	PIs	Users
Center for Applied Isotope Study	1	1
Center for Computational Quantum Chemistry	3	10
Complex Carbohydrate Research Center	6	28
Georgia Genomics Facility	1	5
Institute of Bioinformatics	1	1
Savannah River Ecology Laboratory	3	9
Skidaway Institute of Oceanography	2	2
Center for Family Research	1	1
Carl Vinson Institute of Government	1	2
	19	59

Program Structure: Control Flow

```
if expression:
    statements
elif expression:
    statements
……
else:
    statements
```

```
if a < 0:
    print “a is negative”
elif a == 0:
    print “a is zero”
else:
    print “a is positive”
```

```
if a < b:
    smaller = a
else:
    smaller = b
```

```
if name != “John”:
    pass                              # do nothing
else:
    print “Hello, John!”
```

Note: Examples in this material are for Python2

Program Structure: Loop

- while:

```
while expression:
    statements
```

E.g. :

```
# s and t are two sequences
i = 0
while i < len(s) and i < len(t):
    x = s[i]
    y = t[i]
    print x + y
    i += 1
```

s = [1, 2, 3, 4] : a list
t = (5, 6, 7, 8) : a tuple

Hi, this is Not Python style!

s = [1 , 2 , 3 , 4] : a list

t = (5 , 6 , 7 , 8) : a tuple

zip

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

- for:

```
for i in seq:
    statements
```

E.g. :

```
# s and t are two sequences
for x, y in zip(s, t):
    print x + y
```

Function: Procedural and Functional Programming

- Function:

```
def functionName (params):
    statements
```

E.g. 1:

```
def f(x, y=0):                  # y has a default value of 0
    return x + y                # return a value

f(10, 2)                        # returns 12
f(10)                           # returns 10
```

E.g. 2

```
def f(x, y=0):                  # y has default value of 0
    return (x+y, x–y, x*y, x**y)  # return a tuple

v1, v2, v3, v4 = f(10, 2)       # v1=12, v2=8, v3=20, v4=100
v1, v2, v3, v4 = f(10)          # v1=10, v2=10, v3=0, v4=1
```

Function: Procedural Programming

```
Import sys                                          # load the sys module
def calPrincipal(portfolio):
""" Functions:  1. Read 4-column data line by line from a file: Name, Initial_Principal, Interest_Rate, Years
                2. Calculate final principal for each Name 3. Store 5-column data as a record into a list """
    del portfolio[0:]                               # clear the storing list
    f = open(sys.argv [1], 'r')                     # open a file given as the 1st param. on command line
    for line in f.readlines():                      # read lines; return a list; ending '\n' is also read
        fields = line.split(',')                    # split each line using ',' as a delimiter; return a list
        name = fields[0].strip()                    # remove leading and trailing whitespace
        iniPrincipal = float(fields[1].strip())
        principal = iniPrincipal
        rate = float(fields[2].strip())
        years = int(fields[3].strip('\n'))          # remove leading and trailing whitespace and '\n'

        year = 1
        while year <= years:                        # calculate final principal of 5 years for each Name
            principal =  principal * (1+rate)
            year += 1

        portfolio.append((name, iniPrincipal, rate, years, principal))          # store 5-column record in list
```

principal.txt:

```
Tyler, 2000, 0.05, 5
Mark, 5000, 0.02, 5
Ann, 3000, 0.02, 5
John, 7000, 0.03, 5
```

Function: Procedural Programming

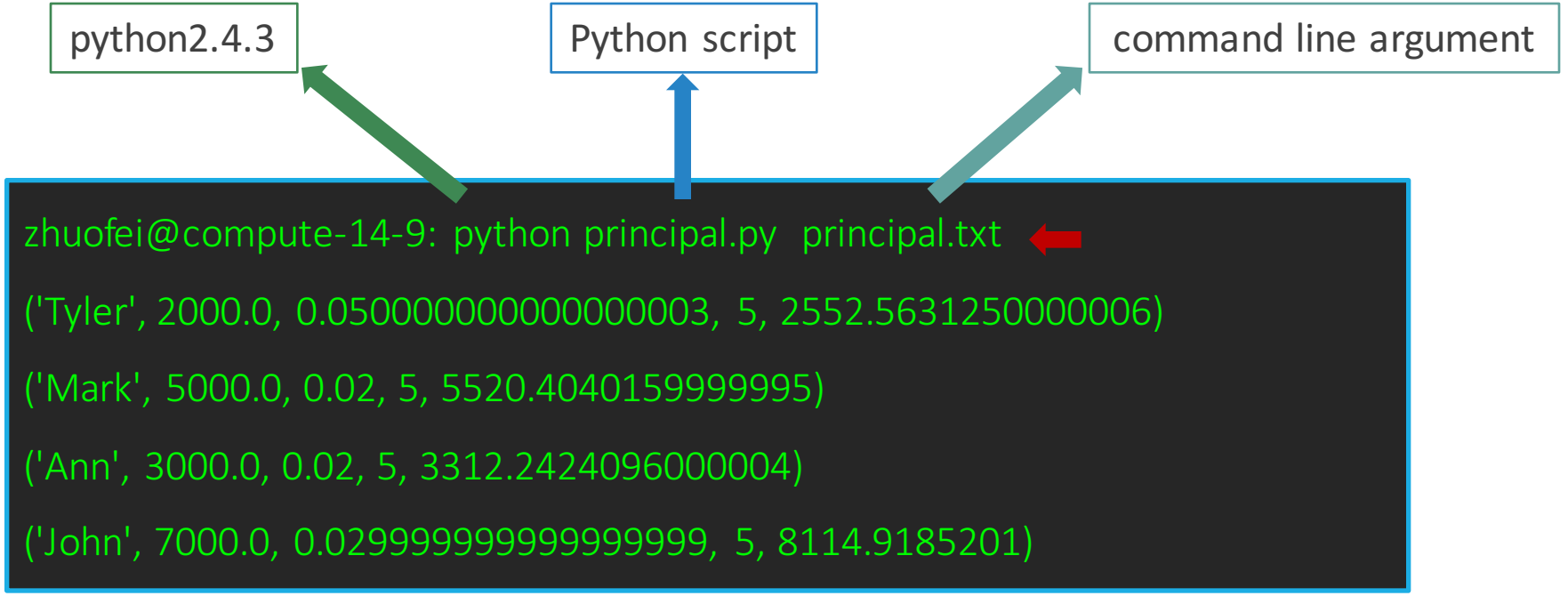
- Calling function (cont.):

```
portfolio = []                          # create a storing list
calPrincipal(portfolio)                 # call function
for t in portfolio: print t             # output to screen; yes, you can put them on the same line
```

Next Page to Run!

Function: Procedural Programming

- Run on zcluster's interactive nodes (qlogin) with default python2.4.3:



Function: Functional Programming

- Functional Programming 101 – function Object: function itself is a data!

```
def square(x):                          # a simple regular function
    return x*x

def g(func):                            # g is taking a function as a parameter, i.e., function itself is a data!
    return func(10)

result = g(square)                      # result = 100
```

- Functional Programming 101 – Decorator: a function wrapper to enhance/alter the behavior of the function object being wrapped

```
def myDecorator(func):                  # wrapper
    print "Hello, I am more human friendly!"
    return func

@myDecorator                            # special @ symbol means square is going to be wrapped by myDecorator
def square(x):
    return x*x

                                        # here I am calling the wrapped square, and output:
                                        #       Hello, I am more human friendly!
print square(10)                        #       100
```

Function: Functional Programming

- Functional Programming 101 – Generator: a function using yield keyword to produce a sequence of values for use in iteration

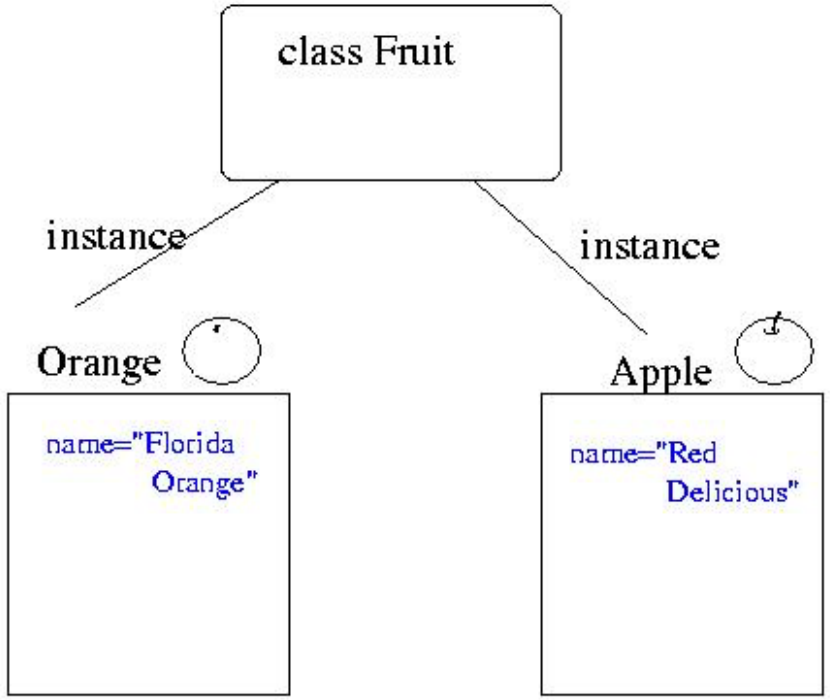
```
def countdown(n):
    while n > 0:
        yield n                    # a function using yield keyword is a generator to produce a value sequence
        n -= 1
    return                         # generator can only return None

c = countdown(10)                  # define a generator object

v1 = c.next()                      # v1 = 10 ; next method of a generator is to produce a value each time it's called
v2 = c.next()                      # v2 = 9
v3 = c.next()                      # v3 = 8

for v in c:                        # normally we use a generator in a for loop
    print v                        # Output:
                                   # 7
                                   # 6
                                   # ……
                                   # 1
```

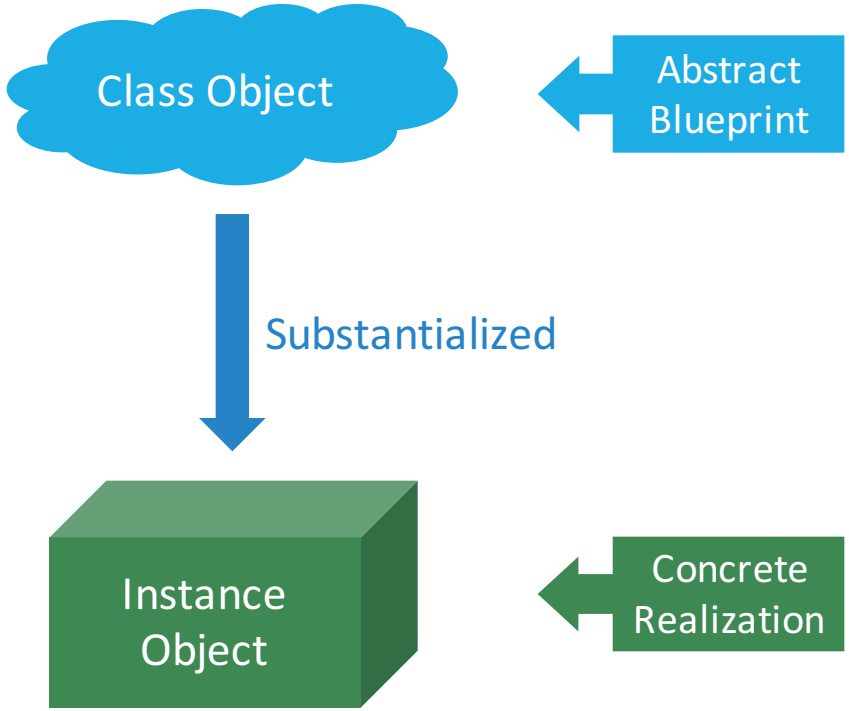
Class: Object-Oriented Programming (OOP)

- Python Class Basics
- Inheritance with Class
- Polymorphism and Class

Python Class Basics

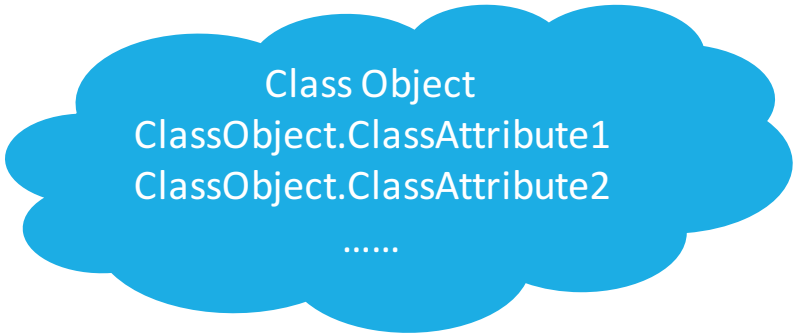
- What are *Class Object* and *Instance Object*?





Python Class Basics

- *Class object* is a Python program blueprint or factory to generate concrete *instance objects*, and support *inheritance* and *polymorphism* of Python OOP
  - Set up a set of class attributes: *class variables, methods, etc.*
  - ClassObject.ClassAttribute to fully specify a class attribute

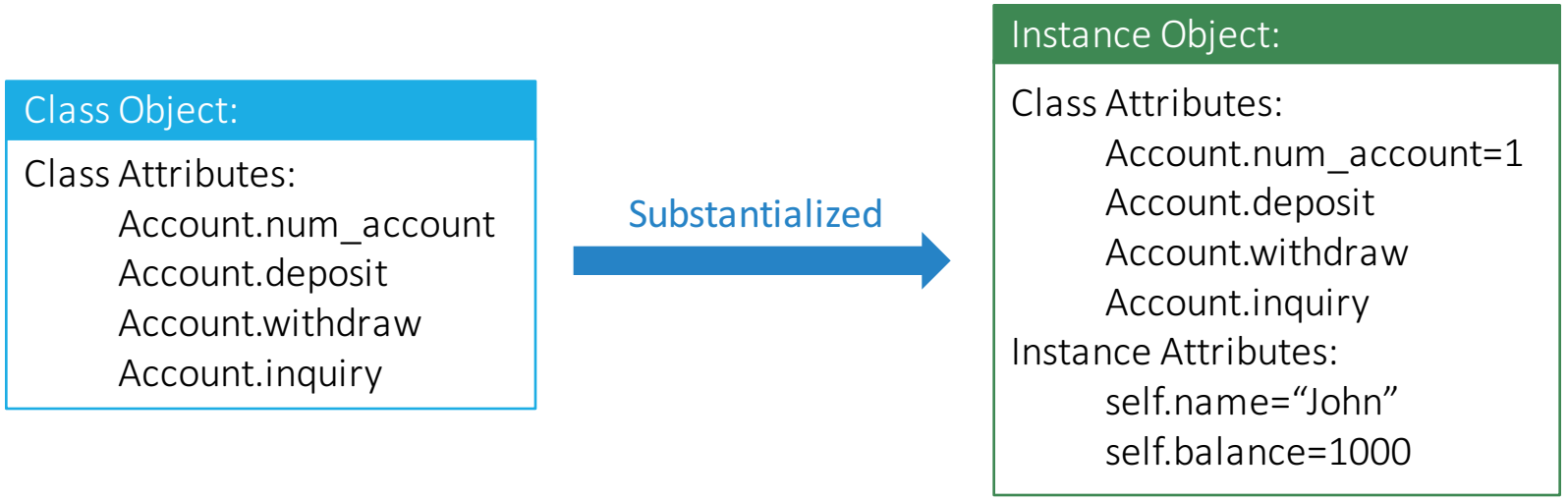


Python Class Basics

- *Instance object* is a *real* and *concrete* object the program processes, generated from a *class object*
  - Set up a set of instance attributes: per-instance attributes
  - InstanceObject.InstanceAttribute to fully specify a instance attribute
  - Class attributes are shared by all instance objects created

Python Class Basics

- Let's try Account!



- To create a class object ➔ `class` statement, e.g., `class Account`

Python Class Basics

- A simple example - Account class object

```
class Account(object):                  # class statement is to create a class object with class attributes: num_account, deposit, withdraw, inquiry

  Account.num_account = 0               # class attribute Account.num_account is initialized

  def __init__(self, name, balance):    # __init__ is used to initialize a instance object, which is referred by self in class definition
    self.name = name                    # instance attribute self.name is initialized
    self.balance = balance              # instance attribute self.balance is initialized
    Account.num_account += 1            # class attribute Account.num_account is modified

  def deposit(self, amount):            # class attribute Account.deposit
    self.balance += amount              # self.balance is modified

  def withdraw(self, amount):           # class attribute Account.withdraw
    self.balance -= amount              # self.balance is modified

  def inquiry(self):                    # class attribute Account.inquiry
    return self.balance                 # self.balance is returned
```

Class Attributes:
- Account.num_account
- Account.deposit
- Account.withdraw
- Account.inquiry

Instance Attributes:
- self.name
- self.balance

Python Class Basics

- To generate a concrete instance object ➔ Calling class object like a function!

```
ins = Account("John",  1000)          # instance object ins has 2 instance attributes and 4 class attributes initialized!

print ins.name + " has a balance of " + str(ins.inquiry())          # use ins.name and call Account.inquiry

ins.deposit(1500)                                                   # call Account.deposit
ins.withdraw(500)                                                   # call Account.withdraw

print ins.name + " has a balance of " + str(ins.inquiry())          # use ins.name and call Account.inquiry
```

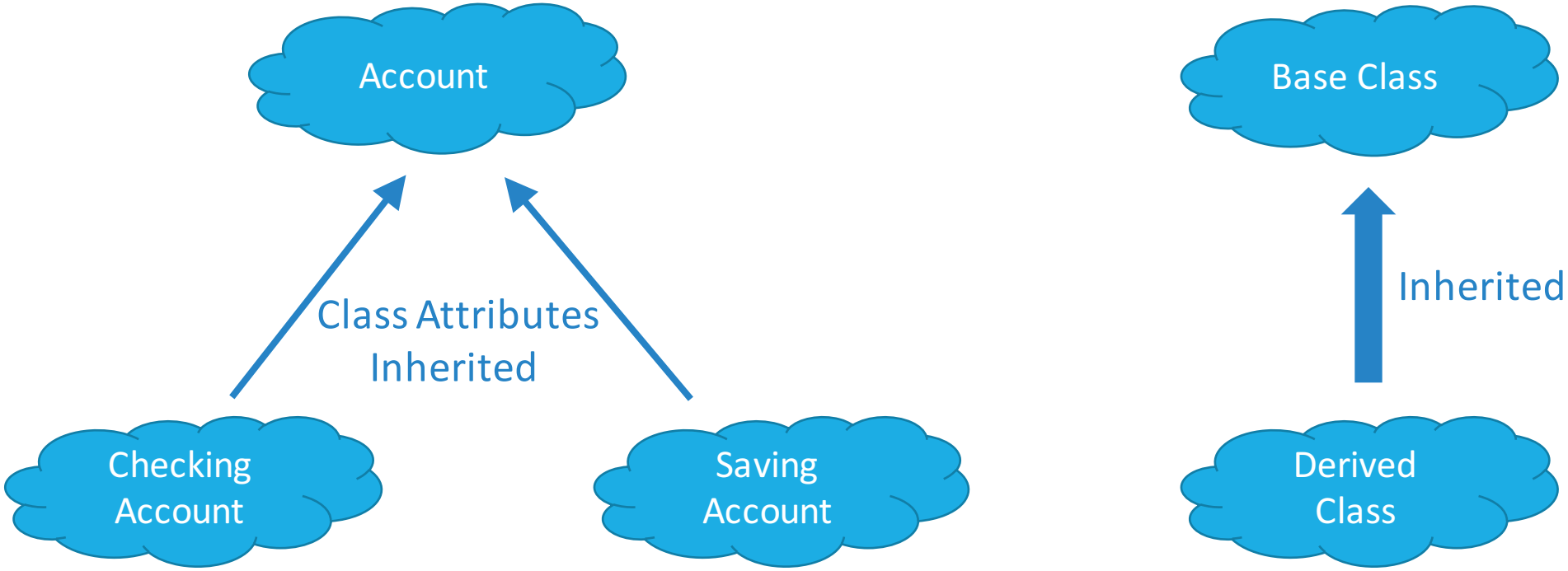
- Output

```
John has a balance of 1000
John has a balance of 2000
```

Call a method with an instance object:
`ins.deposit(500)`

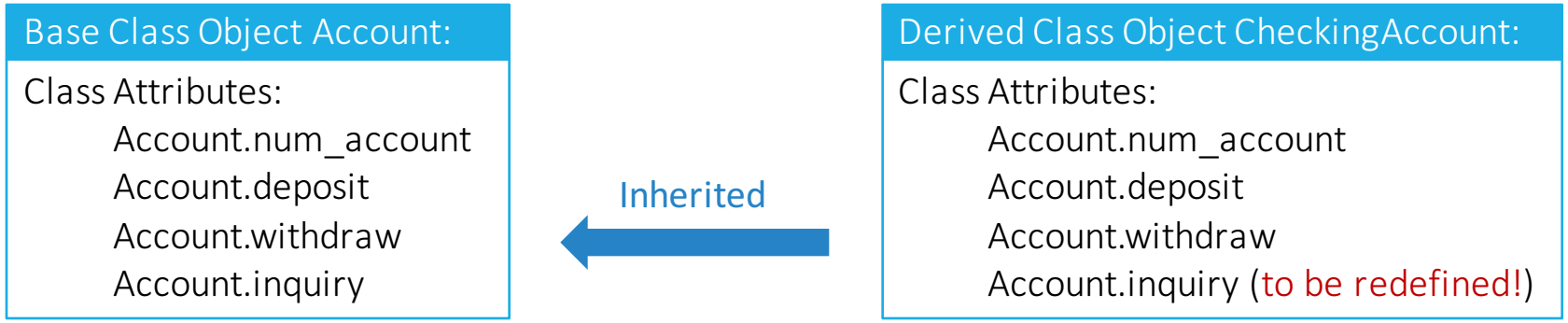
Inheritance with Class

- *What is Inheritance?*



Inheritance with Class

- *Inheritance is a* mechanism for creating a *new class* that redefines or extends the behavior of existing methods of base class ➔ code reuse
- Let's try this!



Inheritance with Class

- A simple example

```
class CheckingAccount(Account):                    # CheckingAccount is inherited from Account: class DerivedClass(BaseClass)

    def __init__(self, name, balance):             # __init__ of the derived class PersonalAccount
        Account.__init__(self,name,balance)        # initialize base class Account by calling Account.__init__()

    def inquiry(self):                             # method from base class is redefined!
        print "Checking Account : " + str(Account.inquiry(self)) + " : " + self.name      # call Account.inquiry inside

ins = CheckingAccount("Peter", 0)                  # instance object ins has 2 instance attributes and 4 class attributes initialized!
ins.deposit(500)                                   # will call Account.deposit
ins.inquiry()                                      # call CheckingAccount.inquiry
```

- Output

```
Checking Account : 500 : Peter
```

Call a method with an instance object:
`ins.deposit(500)`

Polymorphism and Class

- *What is Python Polymorphism?*
  - "Capability to get correct behavior of an instance without knowing its type." [1]
  - "Code shouldn't care about *what an object is*, only about *what it does*!" [2]
  - "Code may be written to work with any kind of object whatsoever as long as it has a certain set of methods." [1]

"when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"

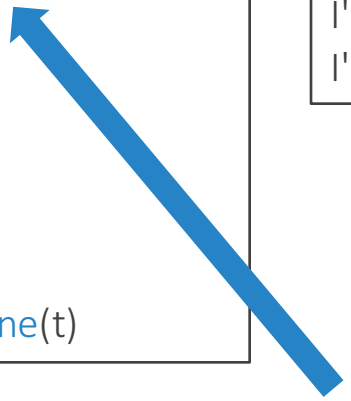
1. *Python Essential Reference, 4th ed.*
2. *Learning Python, 5th ed.*

Polymorphism and Class

- A simple example

```
class DuckThing:
    def quack(self): pass
    def fly(self): pass

class Duck(DuckThing):
    def quack(self):
        print "Quack, quack!"
    def fly(self):
        print "Flap, flap!"

class Person(DuckThing):
    def quack(self):
        print "I'm Quacking!"
    def fly(self):
        print "I'm Flying!"
```

```
def testMachine(unknownThing):
    unknownThing.quack()
    unknownThing.fly()

duck = Duck()
Tom = Person()
testList = [duck, Tom]

for t in testList: testMachine(t)
```



```
Quack, quack!
Flap, flap!
I'm Quacking!
I'm Flying!
```

Polymorphism is here!
Function testMachine works with *any object* as long as it has quack and fly methods!

Thank You!

Let's talk about *Python module and package* on next class!

I : Python introduction, running python, Python built-in data types

II : function (procedural and functional programming) and class (OOP)

III: module, package, and practical code sample