Programming in Python

Sarath Babu

Session-II

Indian Institute of Space Science and Technology
Thiruvananthapuram, Kerala, India 695547

8th August, 2019

IEEE Student Branch IIST

Outline for today

Conditions

if

```
>>> if expression:
        statements
```

if ...else

```
>>> if expression:
        statements
    else:
        statements
```

if ...elif ...else

```
>>> if expression:
        statements
    elif expression:
        statements
    else:
        statements
```

Conditions

if

```
>>> if expression:
        statements
```

if ...else

```
>>> if expression:
        statements
    else:
        statements
```

if ...elif ...else

```
>>> if expression:
        statements
    elif expression:
        statements
    else:
        statements
```

```
>>> a = 2
>>> b = 2
>>> if a == b:
        print('a=b')
```

```
>>> a = 2
>>> b = 4
>>> if a > b:
        print(a)
    else:
        print(b)
```

```
>>> a = 2
>>> b = 4
>>> c = 3
>>> if a > b and a > c:
        print(a)
    elif b > a and b> c:
        print(b)
    else:
        print(c)
```

Loops

while

```
>>> while expression:
        statements
```

for

```
>>> for varibale in sequence:
        statements
```

Loops

while

```
>>> while expression:
        statements
```

```
>>> i = 0
>>> while i <= 10:
        print(i)
        i = i + 2
```

for

```
>>> for varibale in sequence:
        statements
```

```
>>> n = range(0, 11, 2)
>>> for i in n:
        print(i)
>>> for i in range(11):
        print(i)
```

Loops

while

```
>>> while expression:
        statements
```

```
>>> i = 0
>>> while i <= 10:
        print(i)
        i = i + 2
```

for

```
>>> for varibale in sequence:
        statements
```

```
>>> n = range(0, 11, 2)
>>> for i in n:
        print(i)
>>> for i in range(11):
        print(i)
```

break

```
>>> i = 0
>>> while i <= 10:
        if i == 7:
            break
        print(i)
        i = i + 1
```

continue

```
>>> i = 0
>>> while i <= 10:
        if i == 7:
            i += 2
            continue
        print(i)
        i = i + 1
```

Functions

- Method to divide program into reusable modules
- Uses pass-by-reference for arguments

Function Definition

```
>>> def function_name(args):
        statements
        .
        .
        .
    return obj
```

Function Call

```
>>> val = function_name(args)
```

```
>>> def add(a, b):
        c = a + b
    return c

>>> val = add(2, 3)
>>> print(val)
>>> p = 4
>>> q = 3
>>> print(add(p, q))
```

Exceptions

- Unexpected behavior during a program execution
- On exception, Python script stops execution
- Handled using **try** ... **except** ... **finally** statements
  - **try:** code with the chance of exception
  - **except:** the code for handling the exception
  - **finally:** code that executes irrespective of exception

Exceptions

- Unexpected behavior during a program execution
- On exception, Python script stops execution
- Handled using **try ... except ... finally** statements
  - **try:** code with the chance of exception
  - **except:** the code for handling the exception
  - **finally:** code that executes irrespective of exception

---

```
>>> a = int(input('Enter a: '))
>>> b = int(input('Enter b: '))
>>> try:
        c = a / b # Code prone to exception
    except ZeroDivisionError:
        print('Division with 0') # Executes only on exception
    finally:
        print('Program ended') # Code always work
```

> *"If debugging is the process of removing software bugs, then programming must be the process of putting them in."* – **Edsger Dijkstra**

Files

- Way of storing data in permanent storage
- File operations
  1. open
  2. read/write
  3. close

Opening a file

```
>>> file_ptr = open(filename, mode)
```

File modes	
r	Read mode
w	Write mode (New file will be created if the file does not exists. If the file already exists, it will be overwritten)
a	Append mode (New file will be created if the file does not exists. If file already exists, the data is appended to the file)
r+	Reading + writing
Other modes: rb, rb+, wb, w+, wb+, ab, ab+	

Files cont'd

Reading from a file

1. >>> file_ptr.read(size)
2. >>> file_ptr.readline(size)
3. >>> file_ptr.readlines()
4. Using for loop

data.txt

```
1 Alice
2 Bob
3 Eve
4 John
```

Files cont'd

Reading from a file

1. >>> file_ptr.read(size)
2. >>> file_ptr.readline(size)
3. >>> file_ptr.readlines()
4. Using for loop

data.txt

```
1 Alice
2 Bob
3 Eve
4 John
```

```
>>> fp = open('data.txt', 'r')
>>> while True:
        s = fp.read(10)
        print(s)
        if not s:
            break
>>> fp.close()
```

Files cont'd

Reading from a file

1. >>> file_ptr.read(size)
2. >>> file_ptr.readline(size)
3. >>> file_ptr.readlines()
4. Using for loop

data.txt

```
1 Alice
2 Bob
3 Eve
4 John
```

```
>>> fp = open('data.txt', 'r')
>>> while True:
        s = fp.read(10)
        print(s)
        if not s:
            break
>>> fp.close()
```

```
>>> fp = open('data.txt', 'r')
>>> for line in fp:
        print(line)
>>> fp.close()
```

Files cont'd

Reading from a file

1. >>> file_ptr.read(size)
2. >>> file_ptr.readline(size)
3. >>> file_ptr.readlines()
4. Using for loop

data.txt

```
1 Alice
2 Bob
3 Eve
4 John
```

```
>>> fp = open('data.txt', 'r')
>>> while True:
        s = fp.read(10)
        print(s)
        if not s:
            break
>>> fp.close()
```

```
>>> fp = open('data.txt', 'r')
>>> for line in fp:
        print(line)
>>> fp.close()
```

```
>>> fp = open('data.txt', 'r')
>>> lines = fp.readlines()
>>> fp.close()
>>> print(lines)
```

Files cont'd

Writing to a file

```
>>> file_ptr.write(string)
```

```
>>> fp = open('data.txt', 'w')
>>> fp.write('5 Miller')
>>> fp.close()
```

data.txt

```
5 Miller
```

Closing a file

```
>>> file_ptr.close()
```

```
>>> fp = open('data.txt', 'a')
>>> fp.write('5 Miller')
>>> fp.close()
```

data.txt

```
1 Alice
2 Bob
3 Eve
4 John
5 Miller
```

"Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches." – **Paul Graham**

Object oriented thinking

- World can be considered as collection of **objects**
- **Object** $\Rightarrow$ **Attributes** + **Functions**
- Properties of objects
  - Encapsulation
  - Polymorphism
  - Inheritance
  - Abstraction

Object oriented thinking

- World can be considered as collection of **objects**
- **Object** $\Rightarrow$ **Attributes** + **Functions**
- Properties of objects
  - Encapsulation
  - Polymorphism
  - Inheritance
  - Abstraction




Figure 1: Real-world objects

How to realize objects in Python?

- Objects are defined using the keyword **class**
- Definition can be visualized as the mould for creating objects
- Class definition consists of
  1. Attributes (Data members)
  2. Functions (Methods)

How to realize objects in Python?

- Objects are defined using the keyword **class**
- Definition can be visualized as the mould for creating objects
- Class definition consists of
  1. Attributes (Data members)
  2. Functions (Methods)

Object Definition

```
>>> class ClassName:
        Data members
        .
        .
        .
        Method definitions
```

Object Creation

```
>>> object = ClassName()
```

How to realize objects in Python?

- Objects are defined using the keyword **class**
- Definition can be visualized as the mould for creating objects
- Class definition consists of
  1. Attributes (Data members)
  2. Functions (Methods)

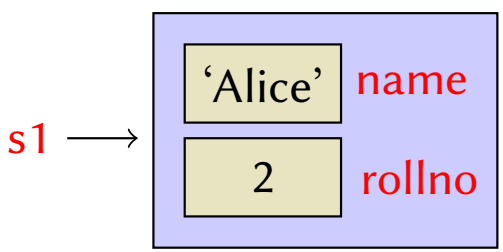
Object Definition

```
>>> class ClassName:
        Data members
        .
        .
        .
        Method definitions
```

Object Creation

```
>>> object = ClassName()
```

```
>>> class Student:
        def __init__(self):
            self.rollno = None
            self.name = None
>>> s1 = Student()
>>> s1.rollno = 2
>>> s1.name = 'Alice'
```



Constructor and methods

Constructor

- Method (or function) used to initialize objects
- Default name is __init__(self,...)

Method

- Function associated with an object
- First argument is always self (represents the calling object)

Constructor and methods

Constructor

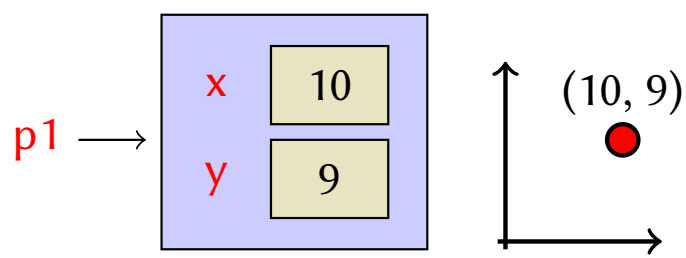
- Method (or function) used to initialize objects
- Default name is __init__(self,...)

Method

- Function associated with an object
- First argument is always self (represents the calling object)

```
>>> class Point2D:
        def __init__(self, x, y):
            self.x = x
            self.y = y
        def display(self):
            print ( '(%f, %f)' % (self.x, self.y) )
        def xscale(self, k):
            self.x = self.x * k
        def yscale(self, k):
            self.y = self.y * k
```

```
>>> p1 = Point2D(2, 3)
>>> p1.xscale(5)
>>> p1.display()
>>> p1.yscale(3)
>>> p1.display()
```



Class variable

- Variable **shared** by objects of a class
- Keyword self is **not** required
- *Modified using class name*
- Accessed using both class name and objects

```
>>> class Point2D:
        pointCount = 0 # Class variable
        def __init__(self, x, y):
            self.x = x
            self.y = y
        def display(self):
            print( '(%f, %f)' % (self.x, self.y) )
        def xscale(self, k):
            self.x = self.x * k
        def yscale(self, k):
            self.y = self.y * k
```

```
>>> p1 = Point2D(2, 3)
>>> Point2D.pointCount += 1
>>> p2 = Point2D(1, 7)
>>> Point2D.pointCount += 1
>>> print (p1.pointCount)
>>> p3 = Point2D(4, 8)
>>> Point2D.pointCount += 1
>>> print (p1.pointCount)
>>> print (p3.pointCount)
>>> print (Point2D.pointCount)
```

Inheritance

- Passing attributes/behavior from parent to offspring
- A class is derived (child, subclass) from existing class/classes (parent, base class)
- Key concept in **code reusability**
- Enables to add additional features **without modifying** existing class/classes
- Reduces the effort in coding

Inheritance

- Passing attributes/behavior from parent to offspring
- A class is derived (child, subclass) from existing class/classes (parent, base class)
- Key concept in **code reusability**
- Enables to add additional features **without modifying** existing class/classes
- Reduces the effort in coding

Syntax

```
>>> class DerivedClass(ParentClass):
        Attribute definitions
              ⋮
        Method definitions
```

Inheritance: 3D Point from 2D Point

```
>>> class Point2D:  # Base class
        def __init__(self, x, y):
            self.x = x
            self.y = y
        def display(self):
            print( '(%f, %f)' % (self.x, self.y) )
        def xscale(self, k):
            self.x = self.x * k
        def yscale(self, k):
            self.y = self.y * k
```

Inheritance: 3D Point from 2D Point

```
>>> class Point2D:  # Base class
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def display(self):
            print( '(%f, %f)' % (self.x, self.y) )

        def xscale(self, k):
            self.x = self.x * k

        def yscale(self, k):
            self.y = self.y * k

>>> class Point3D(Point2D):  # Derived class
        def __init__(self, x, y, z):
            Point2D.__init__(self, x, y)
            self.z = z

        def display(self): # Method overriding
            print( '(%f, %f, %f)' % \
                        (self.x, self.y, self.z) )

        def zscale(self, k):
            self.z = self.z * k
```

Inheritance: 3D Point from 2D Point

```
>>> class Point2D:  # Base class
        def __init__(self, x, y):
            self.x = x
            self.y = y
        def display(self):
            print( '(%f, %f)' % (self.x, self.y) )
        def xscale(self, k):
            self.x = self.x * k
        def yscale(self, k):
            self.y = self.y * k

>>> class Point3D(Point2D):  # Derived class
        def __init__(self, x, y, z):
            Point2D.__init__(self, x, y)
            self.z = z
        def display(self): # Method overriding
            print( '(%f, %f, %f)' % \
                        (self.x, self.y, self.z) )
        def zscale(self, k):
            self.z = self.z * k
```

```
>>> ob1 = Point2D(1, 10)
>>> ob2 = Point3D(4, 5, 6)
>>> ob1.xscale(6)
>>> ob2.xscale(4)
>>> ob1.yscale(2)
>>> ob2.yscale(3)
>>> ob2.zscale(10)
>>> ob1.display()
>>> ob2.display()
```

Polymorphism

- Same name with different meaning
- 'name' implies operator or method
  1. Operator overloading
  2. Function overloading

Polymorphism

- Same name with different meaning
- 'name' implies operator or method
  1. Operator overloading
  2. Function overloading

Operator overloading

```
>>> p1 = Point2D(2, 3)
>>> p2 = Point2D(1, 4)
```

Polymorphism

- Same name with different meaning
- 'name' implies operator or method
  1. Operator overloading
  2. Function overloading

Operator overloading

```
>>> p1 = Point2D(2, 3)
>>> p2 = Point2D(1, 4)
```

Is it possible?

```
>>> p3 = p1 + p2
```

Polymorphism

- Same name with different meaning
- 'name' implies operator or method
  1. Operator overloading
  2. Function overloading

Operator overloading

```
>>> p1 = Point2D(2, 3)
>>> p2 = Point2D(1, 4)
```

Is it possible?

```
>>> p3 = p1 + p2
```

```
>>> class Point2D:
        ⋮
    def __add__(self, p): # Definition for +
        new_x = self.x + p.x
        new_y = self.y + p.y
        q = Point2D(new_x, new_y)
        return q

>>> p1 = Point2D(2, 3)
>>> p2 = Point2D(1, 4)
>>> p3 = p1 + p2  # p3 = p1.__add__(p2)
```

Polymorphic functions

- Functions that execute irrespective of the type of its input

If all of the operations inside the function can be applied to the type, the function can be applied to the type.[a]

[a] Jeffrey Elkner, Allen B Downey, and Chris Meyers. *How to Think Like a Computer Scientist, Learning with Python*. 2002.

Polymorphic functions

- Functions that execute irrespective of the type of its input

> If all of the operations inside the function can be applied to the type, the function can be applied to the type.[a]

[a] Jeffrey Elkner, Allen B Downey, and Chris Meyers. *How to Think Like a Computer Scientist, Learning with Python*. 2002.

Polymorphic?

```
>>> def add_str(a, b):
        p = str(a) + str(b)
        return p
```

Polymorphic functions

- Functions that execute irrespective of the type of its input

If all of the operations inside the function can be applied to the type, the function can be applied to the type.[a]

[a] Jeffrey Elkner, Allen B Downey, and Chris Meyers. *How to Think Like a Computer Scientist, Learning with Python*. 2002.

Polymorphic?

```
>>> def add_str(a, b):
        p = str(a) + str(b)
        return p
```

Polymorphic?

```
>>> def add_str(a, b):
        p = a + b
        return str(p)
```

Data abstraction

- Restricting the data member access
- Only methods can access or modify the data member
- Names of data members start with __

Data abstraction

- Restricting the data member access
- Only methods can access or modify the data member
- Names of data members start with __

```
>>> class Point2D:
        def __init__(self, x, y):
            self.__x = x
            self.y = y
        def display(self):
            print( '(%f, %f)' % (self.__x, self.y) )
        def xscale(self, k):
            self.__x = self.__x * k
        def yscale(self, k):
            self.y = self.y * k
```

Data abstraction

- Restricting the data member access
- Only methods can access or modify the data member
- Names of data members start with __

```
>>> class Point2D:
        def __init__(self, x, y):
            self.__x = x
            self.y = y
        def display(self):
            print( '(%f, %f)' % (self.__x, self.y) )
        def xscale(self, k):
            self.__x = self.__x * k
        def yscale(self, k):
            self.y = self.y * k
```

```
>>> p1 = Point2D()
>>> p1.__x = p1.__x * 3
>>> p1.y = p1.y * 4
>>> p1.xscale(3)
>>> p1.display()
```

"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle." – **Joe Armstrong**

Creating modules

- Program can be split into functions, defined in separate files
- Easy maintenance
- Functions made available using import statement

Creating modules

- Program can be split into functions, defined in separate files
- Easy maintenance
- Functions made available using import statement

---

calc.py

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Creating modules

- Program can be split into functions, defined in separate files
- Easy maintenance
- Functions made available using import statement

---

calc.py

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

```
>>> import calc
>>> a = calc.add(2, 3)
>>> b = calc.subtract(5, 2)
>>> print (a, b)

>>> import calc as cl
>>> a = cl.add(2, 3)
>>> b = cl.subtract(5, 2)
>>> print (a, b)

>>> from calc import *
>>> a = add(2, 3)
>>> b = subtract(5, 2)
>>> print (a, b)
```

Questions?

sarath.babu.2014@ieee.org

Thank you.