# Data Structure and Lab (CSE123)

## Abstract Data Types

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

**Note:** These notes are prepared from the following resources.

- Starting Out with Python, Pearson by Tony Gaddis (2021)
- Introduction to Programming Using Python, Pearson by Y. Daniel Liang, .
- https://docs.oracle.com/javase/tutorial/ (tutorials, and references).
- https://www3.ntu.edu.sg/home/ehchua/programming/index.html#Java
- https://docs.python.org/3/tutorial/

# Last Week

## What is data organization?

- Computer memory stores value of a particular type
- Organizing items: examples from the real life

- Data types
  - Book
  - Shoes
  - Key
  - Necklace
  - Bookshelf
  - Shoes-case
  - Key-holder
  - Necklace-case

YOU HAVE BRAINS IN YOUR HEAD.
YOU HAVE FEET IN YOUR SHOES.
YOU CAN STEER YOURSELF
ANY DIRECTION YOU CHOOSE.

# Last Week (cont...)

## Data Structures

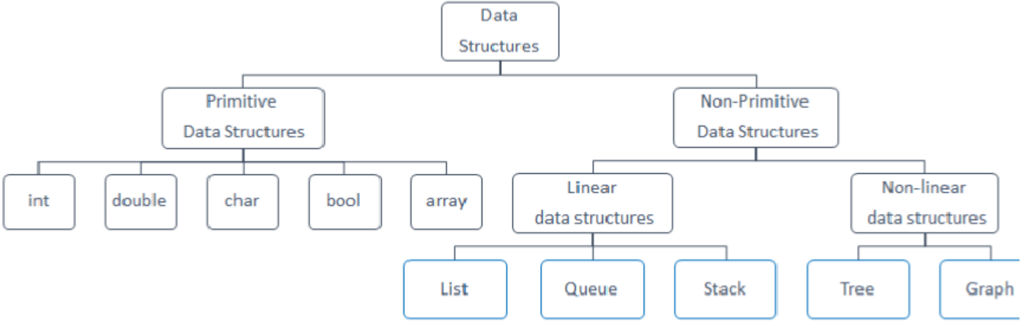


Figure 1: Classification of Data structures

# Contents

## 1 Python Functions

- Function Parameters and Arguments
- Python Anonymous/Lambda Function

## 2 Object Oriented Programming Concept (OOP) in Python

- Python Objects and Classes
- Defining a Class in Python
- Class Variables and Instance Variables
- Instance Methods, Class Methods and Static Methods
- Private/Protected Variables and Methods?
- Operator/Method Overloading

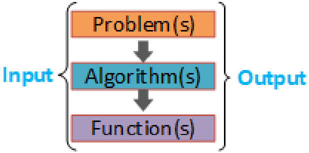
## 3 Examples

- Point class
- Circle class
- Complex Numbers
- The Bag Abstract Data Type

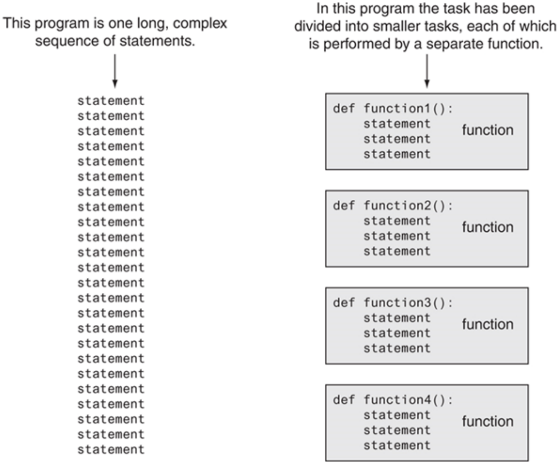
# Python Functions

## *Python Functions*

- A function is a set of statements that takes input, do some specific computation and produces output.



- Functions break larger program into smaller and modular chunks of codes. As our program grows larger and larger, functions make it more organized and manageable.
- Functions allow us to define reusable blocks of code that can be used repeatedly in a program.


This program is one long, complex sequence of statements.

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

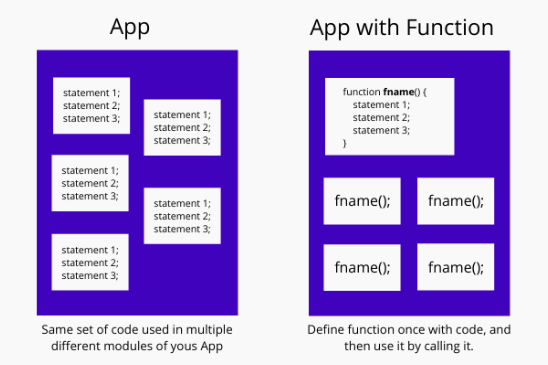
In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():
    statement
    statement
    statement

def function2():
    statement
    statement
    statement

def function3():
    statement
    statement
    statement

def function4():
    statement
    statement
    statement
```

# Python Functions (cont...)

## Why Use Functions?

- Maximizing code reuse and minimizing redundancy: Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time.



```
statement 1;
statement 2;
statement 3;
```

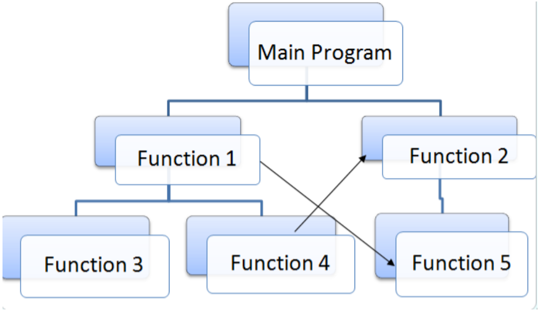
```
function fname() {
    statement 1;
    statement 2;
    statement 3;
}
```

```
fname();
```

Same set of code used in multiple different modules of yous App

Define function once with code, and then use it by calling it.

- Procedural decomposition: Functions also provide a tool for splitting systems into pieces that have well-defined roles.

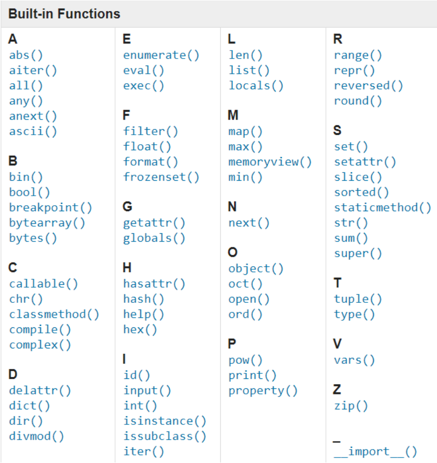


- Types pf functions: 1) **Built-in functions** - Functions that are built into Python (*print(), len() , type()*). 2) **User-defined functions** - Functions defined by the users themselves. 3)**lambda functions**

# Built-in Functions

## *Built-in Functions*

- The Python interpreter has a number of functions and types built into it that are always available.



**Built-in Functions**

**A**
abs()
aiter()
all()
any()
anext()
ascii()

**B**
bin()
bool()
breakpoint()
bytearray()
bytes()

**C**
callable()
chr()
classmethod()
compile()
complex()

**D**
delattr()
dict()
dir()
divmod()

**E**
enumerate()
eval()
exec()

**F**
filter()
float()
format()
frozenset()

**G**
getattr()
globals()

**H**
hasattr()
hash()
help()
hex()

**I**
id()
input()
int()
isinstance()
issubclass()
iter()

**L**
len()
list()
locals()

**M**
map()
max()
memoryview()
min()

**N**
next()

**O**
object()
oct()
open()
ord()

**P**
pow()
print()
property()

**R**
range()
repr()
reversed()
round()

**S**
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

**T**
tuple()
type()

**V**
vars()

**Z**
zip()

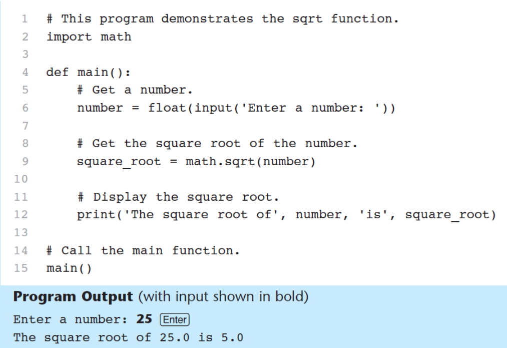
**_**
__import__()

- Examples

```
dir()   # show the names in the module namespace
class Shape:
def __dir__(self):
return ['area', 'perimeter', 'location']
s = Shape()
dir(s) #['area', 'location', 'perimeter']

bin(3) #'0b11' Converts an integer number to a binary string prefixed with "0b"
hex(255) # '0xff' Convert an integer number to a lowercase hexadecimal string prefixed with "0x".
oct(8)  # '0o10' Convert an integer number to an octal string prefixed with "0o".
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
list(enumerate(seasons))
#[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
list(enumerate(seasons, start=1))
#[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

def any(iterable):
for element in iterable:
if element:
return True
return False

class C(B):
def method(self, arg):
   super().method(arg)
# This does the same thing as:
# super(C, self).method(arg)
```

# Built-in Functions (cont...)

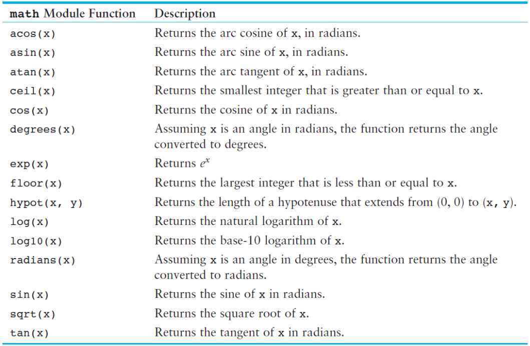
- The math Module: The Python standard library's math module contains numerous functions that can be used in mathematical calculations.
- The math module also defines two variables, pi and e, which are assigned mathematical values for pi and e.



```
# This program demonstrates the sqrt function.
import math

def main():
    # Get a number.
    number = float(input('Enter a number: '))

    # Get the square root of the number.
    square_root = math.sqrt(number)

    # Display the square root.
    print('The square root of', number, 'is', square_root)

# Call the main function.
main()
```

**Program Output** (with input shown in bold)

```
Enter a number: 25 Enter
The square root of 25.0 is 5.0
```



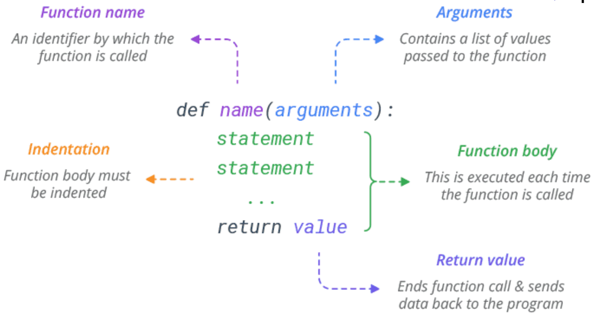
| `math` Module Function | Description |
|---|---|
| `acos(x)` | Returns the arc cosine of `x`, in radians. |
| `asin(x)` | Returns the arc sine of `x`, in radians. |
| `atan(x)` | Returns the arc tangent of `x`, in radians. |
| `ceil(x)` | Returns the smallest integer that is greater than or equal to `x`. |
| `cos(x)` | Returns the cosine of `x` in radians. |
| `degrees(x)` | Assuming `x` is an angle in radians, the function returns the angle converted to degrees. |
| `exp(x)` | Returns $e^x$ |
| `floor(x)` | Returns the largest integer that is less than or equal to `x`. |
| `hypot(x, y)` | Returns the length of a hypotenuse that extends from (0, 0) to (`x`, `y`). |
| `log(x)` | Returns the natural logarithm of `x`. |
| `log10(x)` | Returns the base-10 logarithm of `x`. |
| `radians(x)` | Assuming `x` is an angle in degrees, the function returns the angle converted to radians. |
| `sin(x)` | Returns the sine of `x` in radians. |
| `sqrt(x)` | Returns the square root of `x`. |
| `tan(x)` | Returns the tangent of `x` in radians. |

# User defined Functions

## *User defined Functions*

- In Python, you define a function via the keyword def followed by the function name, the parameter list, the doc-string and the function body. Inside the function body, you can use a return statement to return a value to the caller. There is no need for type declaration like C/C++/Java.

- Defining a Function: To define a Python function def keyword is used. The basic syntax for a Python function definition is shown:



Function name

An identifier by which the function is called

Arguments

Contains a list of values passed to the function

```
def name(arguments):
    statement
    statement
       ...
    return value
```

Indentation

Function body must be indented

Function body

This is executed each time the function is called

Return value

Ends function call & sends data back to the program

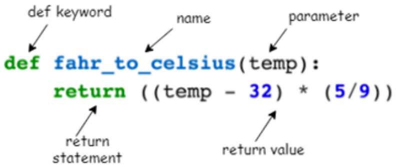
- Function's components.

1. Keyword def that marks the start of the function.
2. A function name to uniquely identify the function.
3. Arguments(optional) through which we pass values to a function.
4. A colon (:) to mark the end of the function header.
5. docstring (optional): A documentation string to describe what the function does.
6. BodyOne or more valid python statements that make up the function body.
7. return statement(optional) to return a value from the function.

# User defined Functions (cont...)

- Example 1: Let's define a function fahr_to_celsius that converts temperatures from Fahrenheit to Celsius:

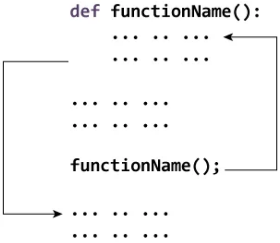
```python
def fahr_to_celsius(temp):
    """
    This function converts temperatures
    from Fahrenheit to Celsius
    """

    return ((temp - 32) * (5/9))

f=fahr_to_celsius(28)
print('freezing point of water:', fahr_to_celsius(32), 'C')
print('boiling point of water:', fahr_to_celsius(212), 'C')
```



```python
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
```

- Example 2

```python
# Define a function (need to define before using the function)
def fibon(n):
    """Print the first n Fibonacci numbers, where f(n)=f(n-1)+f(n-2) and f(1)=f(2)=1"""
    a, b = 1, 1   # pair-wise assignment
    for count in range(n): # count = 0, 1, 2, ..., n-1
    print(a, end=' ')  # print a space instead of a default newline at the end
    a, b = b, a+b
    print()   # print a newline
```



- Calling a Function: The function is called by adding parentheses after the function's name and providing the value(s) for the argument(s).

# User defined Functions (cont...)

- Function doc-string: A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the __doc__ special attribute of that object.
- String literals occurring elsewhere in Python code may also act as documentation. They are not recognized by the Python bytecode compiler and are not accessible as runtime object attributes.
- Triple quotes are used even though the string fits on one line. This makes it easy to later expand it.
- Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description.

```python
def complex(real=0.0, imag=0.0):
"""Form a complex number.

Keyword arguments:
real -- the real part (default 0.0)
imag -- the imaginary part (default 0.0)
"""
if imag == 0.0 and real == 0.0:
return complex\_zero
...
```

- The pass statement The pass statement does nothing. It is sometimes needed as a dummy statement placeholder to ensure correct syntax, e.g.,

```python
def my_fun():
    pass      # To be defined later, but syntax error if empty
```

# Function Parameters and Arguments

## Passing Arguments by Value vs. by Reference

- Immutable arguments (such as integers, floats, strings and tuples) are passed by value. That is, a copy is cloned and passed into the function. The original cannot be modified inside the function.
- Mutable arguments (such as lists, dictionaries, sets and instances of classes) are passed by reference. That is, they can be modified inside the function.

```
# Immutable argument pass-by-value
def increment_int(number):
number += 1
number = 5
increment_int(number)
print(number)   # no change

# Mutable argument pass-by-reference
def increment_list(lst):
for i in range(len(lst)):
lst[i] += lst[i]
lst = [1, 2, 3, 4, 5]
increment_list(lst)
print(lst) #[2, 4, 6, 8, 10] not thr changed list
```

- Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition.
- Types of Arguments
  - Default Arguments
  - Positional Arguments
  - Keyword Arguments
  - Variable Length Positional Arguments (*args)
  - Variable Length Keyword Arguments (**kwargs)

# Function Parameters and Arguments (cont...)

- Default Arguments: You can assign a default value to the ”trailing” function parameters. These trailing parameters having default values are optional during invocation.
- In stead of hard-coding the 'hello, ', it is more flexible to use a parameter with a default value,

```python
def my_sum(n1, n2 = 4, n3 = 5):  # n1 is required, n2 and n3 having defaults are optional
"""Return the sum of all the arguments"""
return n1 + n2 + n3

print(my_sum(1, 2, 3)) #6
print(my_sum(1, 2))    # 8 n3 defaults 8
print(my_sum(1))       # 10 n2 and n3 default
print(my_sum()) #TypeError: my_sum() takes at least 1 argument (0 given)
print(my_sum(1, 2, 3, 4)) #TypeError: my_sum() takes at most 3 arguments (4 given)
```

```python
def greet(name):
return 'hello, ' + name
greet('Peter')  # Output: 'hello, Peter'

def greet(name, prefix='hello'):  # 'name' is required, 'prefix' is optional
return prefix + ', ' + name
greet('Peter')                    # Output: 'hello, Peter'
greet('Peter', 'hi')              # Output: 'hi, Peter'
greet('Peter', prefix='hi')       # Output: 'hi, Peter'
greet(name='Peter', prefix='hi')  # Output: 'hi, Peter'
```

- Positional Arguments: When we call a function with some values, these values get assigned to the arguments according to their position.
- The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.

```python

greet("Monica", "Good morning!") # 2 Positional Arguments

greet( "Good morning!", "Monica") # 2 Positional Arguments (out of order)

greet("Monica") # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'

greet()    # no arguments
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

# Function Parameters and Arguments (cont...)

- **Keyword Arguments:** Python allows functions to be called using keyword arguments. A keyword argument is an argument passed to a function or method which is preceded by a keyword and an equals sign. When we call functions in this way, the order (position) of the arguments can be changed. You can also mix the positional arguments and keyword arguments,

```python
def my_sum(n1, n2 = 4, n3 = 5):
    """Return the sum of all the arguments"""
    return n1 + n2 + n3

print(my_sum(n2 = 2, n1 = 1, n3 = 3)) # Keyword arguments need not follow their positional order
print(my_sum(n2 = 2, n1 = 1))         # n3 defaults
print(my_sum(n1 = 1))                 # n2 and n3 default
print(my_sum(1, n3 = 3))              # n2 default. Place positional arguments before keyword arguments
print(my_sum(n2 = 2))                 # TypeError, n1 missing
```

```python
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

#1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")

#SyntaxError: non-keyword arg after keyword arg
greet(name="Bruce","How do you do?")
```

- **Variable Number of Positional Parameters (*args):** Python supports variable (arbitrary) number of arguments. In the function definition, you can use * to pack all the remaining positional arguments into a tuple.

# Function Parameters and Arguments (cont...)

```
def my_sum(a, *args):  # Accept one positional argument, followed by
       arbitrary number of arguments pack into tuple
"""Return the sum of all the arguments (one or more)"""
sum = a
print('args is:', args)  # for testing
for item in args:  # args is a tuple
sum += item
return sum

print(my_sum(1))            # args is: ()
print(my_sum(1, 2))         # args is: (2,)
print(my_sum(1, 2, 3))      # args is: (2, 3)
print(my_sum(1, 2, 3, 4))   # args is: (2, 3, 4)
```

```
def my_sum(a, *args, b):
sum = a
print('args is:', args)
for item in args:
sum += item
sum += b
return sum

print(my_sum(1, 2, 3, 4)) # TypeError: my_sum() missing 1 required
       keyword-only argument: 'b'
print(my_sum(1, 2, 3, 4, b=5))   # args is: (2, 3, 4)
```

- **Unpacking List/Tuple into Positional Arguments (*lst, *tuple)** In the reverse situation when the arguments are already in a list/tuple, you can also use * to unpack the list/tuple as separate positional arguments. For example,

```
def my_sum(*args):  # Variable number of positional arguments
sum = 0
for item in args: sum += item
return sum
print(my_sum(11, 22, 33))  # positional arguments
lst = [44, 55, 66]
print(my_sum(*lst))   # Unpack the list into positional arguments
tup = (7, 8, 9, 10)
print(my_sum(*tup))   # Unpack the tuple into positional arguments
```

# Function Parameters and Arguments (cont...)

- Variable Number of Keyword Parameters (**kwargs): For keyword parameters, you can use ** to pack them into a dictionary. For example

```
def my_print_kwargs(msg, **kwargs):  # Accept variable number of keyword arguments, pack into dictionary
print(msg)
for key, value in kwargs.items():  # kwargs is a dictionary
print('{}: {}'.format(key, value))

my_print_kwargs('hello', name='Peter', age=24) # hello name: Peter age: 24
```

- Unpacking Dictionary into Keyword Arguments (**dict) Similarly, you can also use ** to unpack a dictionary into individual keyword arguments

```
def my_print_kwargs(msg, **kwargs):  # Accept variable number of keyword arguments, pack into dictionary
print(msg)
for key, value in kwargs.items():  # kwargs is a dictionary
print('{}: {}'.format(key, value))

dict = {'k1':'v1', 'k2':'v2', 'k3':'v3'}
print(my_print_kwargs('hello', **dict))  # Use ** to unpack dictionary into separate keyword arguments k1=v1, k2=v2
```

# Function Parameters and Arguments (cont...)

- **Using both *args and **kwargs:** You can use both *args and **kwargs in your function definition. Place *args before **kwargs. For example,

```
def my_print_all_args(*args, **kwargs):   # Place *args before **kwargs
for item in args:  # args is a tuple
print(item)
for key, value in kwargs.items():  # kwargs is a dictionary
print('%s: %s' % (key, value))

print(my_print_all_args('a', 'b', 'c', name='Peter', age=24))
lst = [1, 2, 3]
dict = {'name': 'peter'}
print(my_print_all_args(*lst, **dict))  # Unpack
```

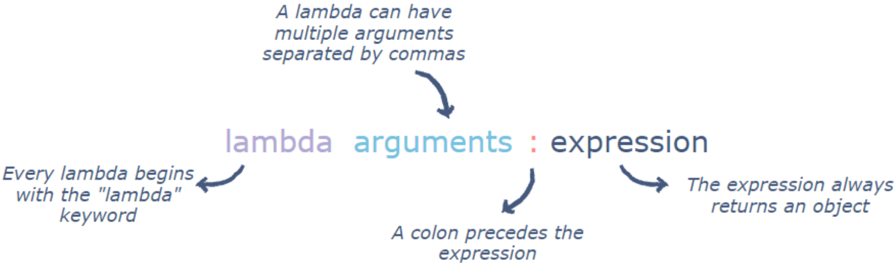
- **Function Return Values:** You can return multiple values from a Python function, e.g.,

```
def my_fun():
return 1, 'a', 'hello'  # Return a tuple

x, y, z = my_fun()  # Chain assignment
print(my_fun())  # Returns a tuple (1, 'a', 'hello')
```

# Python Anonymous/Lambda Function

*Python Anonymous/Lambda Function*

- A lambda function is a small anonymous function which returns an object.
- The object returned by lambda is usually assigned to a variable or used as a part of other bigger functions.
- Instead of the conventional def keyword used for creating functions, a lambda function is defined by using the lambda keyword.
- The structure of lambda can be seen below:



A lambda can have multiple arguments separated by commas

lambda arguments : expression

Every lambda begins with the "lambda" keyword

A colon precedes the expression

The expression always returns an object

# Python Anonymous/Lambda Function (cont...)

- A lambda is much more readable than a full function since it can be written in-line. Hence, it is a good practice to use lambdas when the function expression is small.

- A lambda function can be immediately invoked. For this reason it is often referred to as an Immediately Invoked Function Expression (IIFE).

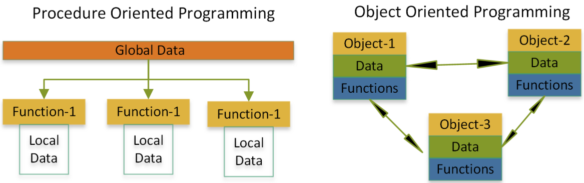
- Lambda functions are used along with built-in functions like filter(), map() etc. Examples of simple lambdas are given here,

```
print((lambda x: x*2)(3))  # Prints 6

square = lambda n : n*n     # A squaring lambda function
num = square(5) print num

# Positional arguments
add = lambda x, y, z: x+y+z
print(add(2, 3, 4))  # Prints 9

# Keyword arguments
add = lambda x, y, z: x+y+z
print(add(2, z=3, y=4)) # Prints 9

# Default arguments
add = lambda x, y=3, z=4: x+y+z
print(add(2))  # Prints 9


# *args
add = lambda *args: sum(args)
print(add(2, 3, 4)) # Prints 9

# **args
add = lambda **kwargs: sum(kwargs.values())
print(add(x=2, y=3, z=4)) # Prints 9


# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)

# Program to double each item in a list using map()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)^^I^^I^^I
```

# (OOP) Concept in Python

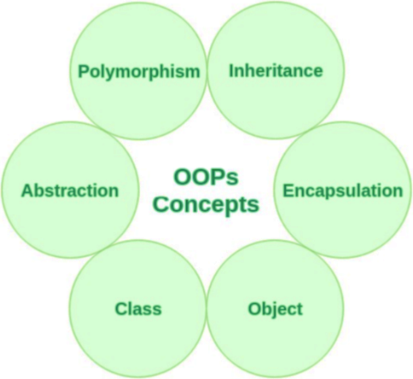
## Object Oriented Programming Concept (OOP) in Python

- Python is an object oriented programming language.
- Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.



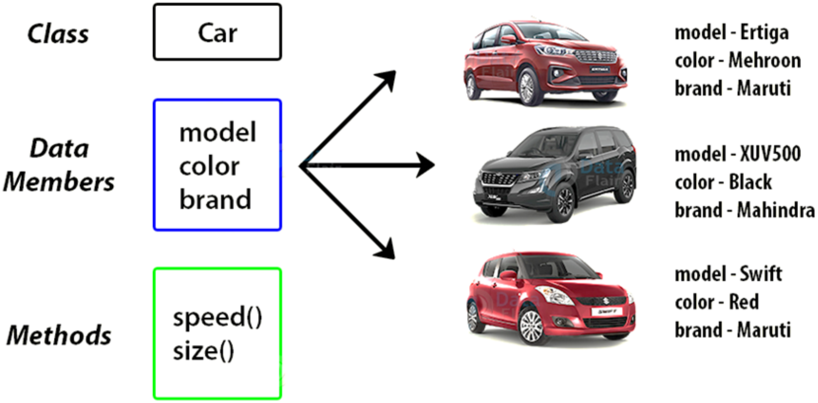
- Objects are created from their classes

- The major features in object-oriented programming that makes them different than non-OOP languages: encapsulation, inheritance and polymorphism.



# (OOP) Concept in Python (cont...)

- An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.



# (OOP) Concept in Python (cont...)

## Defining a Class in Python

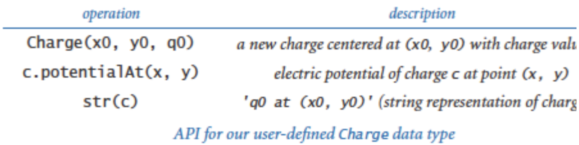
- A class creates a new local namespace where all its members are defined.
- We define a class in Python using “class” keyword.
- The body of the class contains: variables and methods
- Variables (attributes, data members) can be of different types
  - instance variables
  - class variables
- class can contain different types of methods (function members)
  - instance methods (special methods ordinary methods)
  - class methods

- Syntax for creating a class

```
class class_name(super_1,...):


class_var_1 = value_1    # Class variables
......

def __init__(self, arg_1, ...):   # Initializer
self.instance_var_1 = arg_1  #instance variables
......

def __str__(self):   # special method str()"""
......

def __repr__(self):  # special method repr()"""
......

def method_name(self, arg_1, ... ):   # Ordinary method"""
......
```

# (OOP) Concept in Python (cont...)

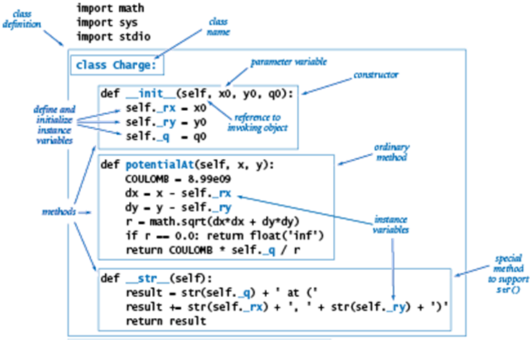
- Example Consider a data type Charge. (Coulomb s law which tells us that the electric potential at a point due to a given charged particle is represented by V = kq/r, where q is the charge value, r is the distance from the point to the charge, and k = 8.99 × 109 N m2/C2 is a constant known as the electrostatic constant, or Coulomb's constant. ).
- Charge data type



| *operation* | *description* |
|---|---|
| `Charge(x0, y0, q0)` | *a new charge centered at (x0, y0) with charge valu* |
| `c.potentialAt(x, y)` | *electric potential of charge c at point (x, y)* |
| `str(c)` | *'q0 at (x0, y0)' (string representation of charg* |

*API for our user-defined Charge data type*

- Python code



```
import math
import sys
import stdio

class Charge:

    def __init__(self, x0, y0, q0):
        self._rx = x0
        self._ry = y0
        self._q  = q0

    def potentialAt(self, x, y):
        COULOMB = 8.99e09
        dx = x - self._rx
        dy = y - self._ry
        r = math.sqrt(dx*dx + dy*dy)
        if r == 0.0: return float('inf')
        return COULOMB * self._q / r

    def __str__(self):
        result = str(self._q) + ' at ('
        result += str(self._rx) + ', ' + str(self._ry) + ')'
        return result
```

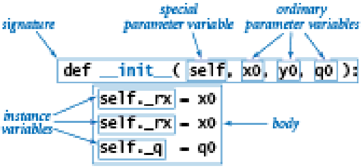
class definition
class name
parameter variable
constructor
define and initialize instance variables
reference to invoking object
ordinary method
methods
instance variables
special method to support str()

Reference *https* : //*introcs.cs.princeton.edu*/*python*/32*class*/

# (OOP) Concept in Python (cont...)

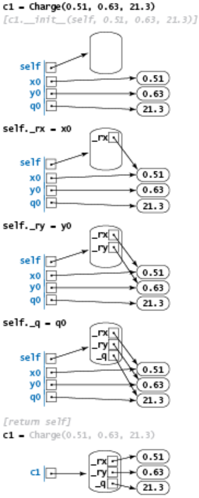
## Creating and initializing an Object

- *self* represents the instance of the class. It can be used to access the attributes and methods of the class in python.
- The __init__ function is a reserved function in classes in Python which is automatically called whenever a new object of the class is instantiated.



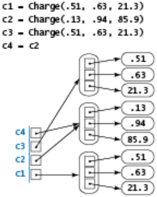
```
def __init__( self, x0, y0, q0 ):
    self._rx = x0
    self._rx = x0
    self._q  = q0
```

Anatomy of a constructor



```
c1 = Charge(0.51, 0.63, 21.3)
[c1.__init__(self, 0.51, 0.63, 21.3)]
self._rx = x0
self._ry = y0
self._q = q0
[return self]
c1 = Charge(0.51, 0.63, 21.3)
```

Creating and initializing an object



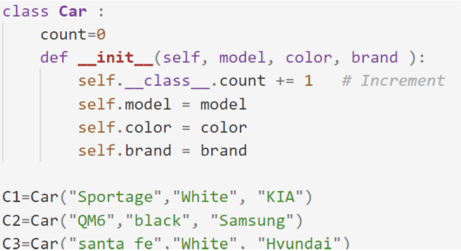
```
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.13, .94, 85.9)
c3 = Charge(.51, .63, 21.3)
c4 = c2
```

Four variables referring to three Charge objects

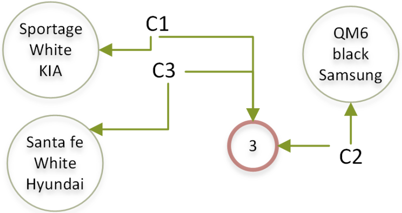
# (OOP) Concept in Python (cont...)

## Class Variables and Instance Variables

- Instance variables are for data unique to each instance (object)
- Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.
- Class Variable: A class variable is nothing but a variable that is defined outside the constructor. A class variable is also called as a static variable.
- **class variables** are shared by all instances of the class.
- In Example: count is a class variable and color, speed are instance variables



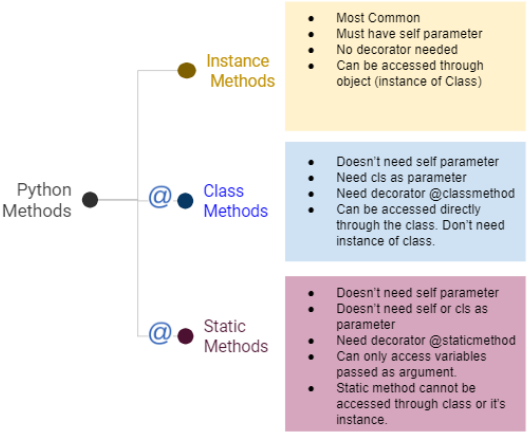
```
class Car :
    count=0
    def __init__(self, model, color, brand ):
        self.__class__.count += 1   # Increment
        self.model = model
        self.color = color
        self.brand = brand

C1=Car("Sportage","White", "KIA")
C2=Car("QM6","black", "Samsung")
C3=Car("santa fe","White", "Hyundai")
```



# (OOP) Concept in Python (cont...)

## Instance Methods, Class Methods and Static Methods

- Instance Method : Instance methods are the most common type of method. An instance method is invoked by an instance object (and not a class object). It takes the instance (self) as its first argument.
- Class Method: A class method belongs to the class and is a function of the class. It is declared with the @classmethod decorator. It accepts the class (cls) as its first argument.
- Static Method : A static method is declared with a @staticmethod decorator. It does not depends on the state of the object and could be a separate function of a module. A static method can be invoked via a class object or instance object.



Python Methods

Instance Methods

- Most Common
- Must have self parameter
- No decorator needed
- Can be accessed through object (instance of Class)

@ Class Methods

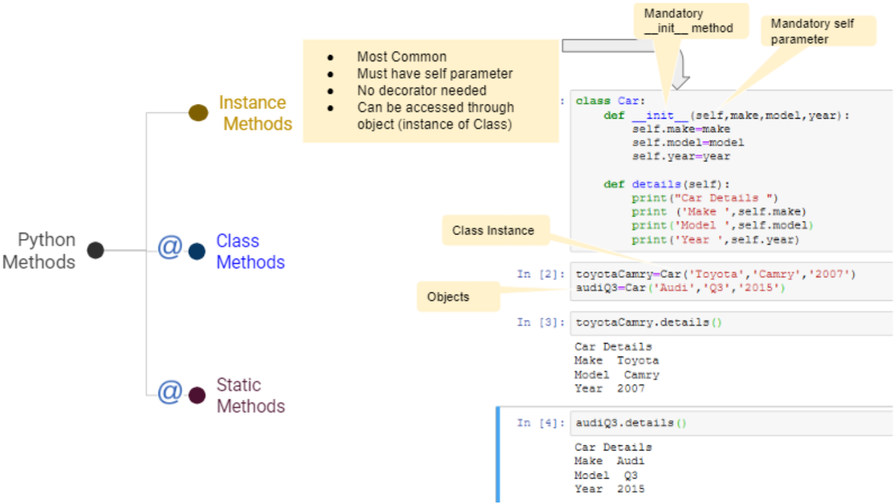
- Doesn't need self parameter
- Need cls as parameter
- Need decorator @classmethod
- Can be accessed directly through the class. Don't need instance of class.

@ Static Methods

- Doesn't need self parameter
- Doesn't need self or cls as parameter
- Need decorator @staticmethod
- Can only access variables passed as argument.
- Static method cannot be accessed through class or it's instance.

- Instance Method Example

# (OOP) Concept in Python (cont...)



- Most Common
- Must have self parameter
- No decorator needed
- Can be accessed through object (instance of Class)

```
:  class Car:
       def __init__(self,make,model,year):
           self.make=make
           self.model=model
           self.year=year

       def details(self):
           print("Car Details ")
           print ('Make ',self.make)
           print('Model ',self.model)
           print('Year ',self.year)
```

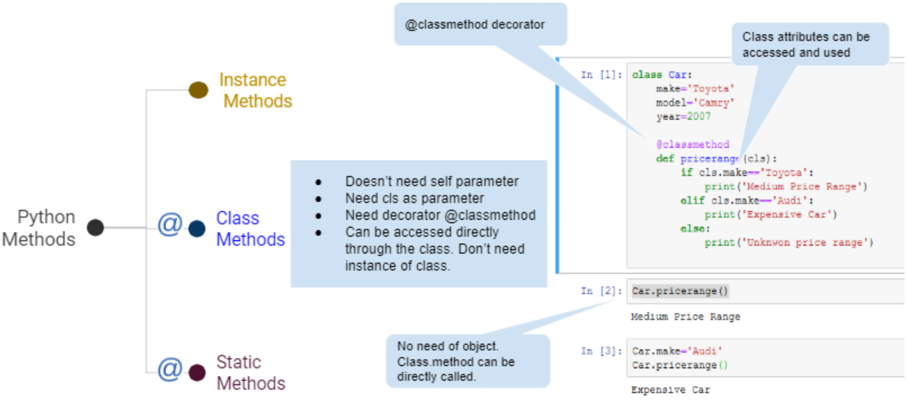
```
In [2]: toyotaCamry=Car('Toyota','Camry','2007')
        audiQ3=Car('Audi','Q3','2015')
```

```
In [3]: toyotaCamry.details()

        Car Details
        Make  Toyota
        Model  Camry
        Year  2007
```

```
In [4]: audiQ3.details()

        Car Details
        Make  Audi
        Model  Q3
        Year  2015
```

- A class method has a decorator. A class method does not need object and can access and manipulate the class attributes directly through the class.

# (OOP) Concept in Python (cont...)



- Doesn't need self parameter
- Need cls as parameter
- Need decorator @classmethod
- Can be accessed directly through the class. Don't need instance of class.

@classmethod decorator

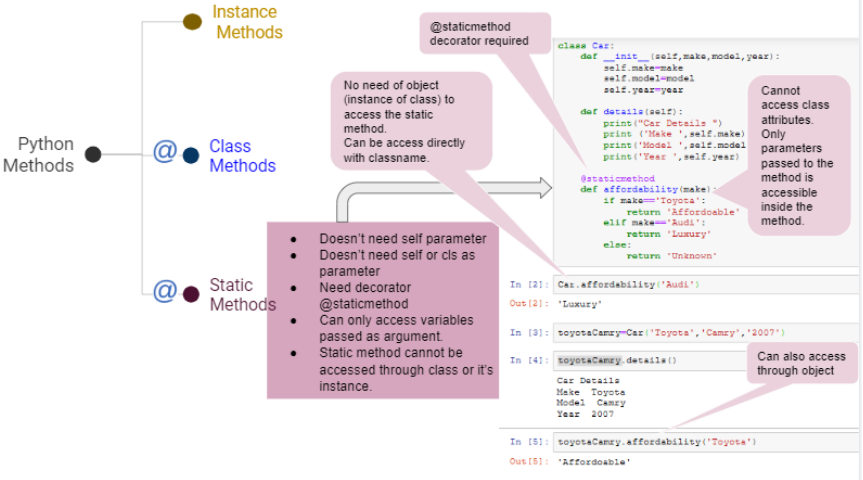
Class attributes can be accessed and used

No need of object. Class.method can be directly called.

```
In [1]: class Car:
            make='Toyota'
            model='Camry'
            year=2007

            @classmethod
            def pricerange(cls):
                if cls.make=='Toyota':
                    print('Medium Price Range')
                elif cls.make=='Audi':
                    print('Expensive Car')
                else:
                    print('Unknwon price range')

In [2]: Car.pricerange()

        Medium Price Range

In [3]: Car.make='Audi'
        Car.pricerange()

        Expensive Car
```

- Like class method, a static method also has a decorator. Unlike the class methods, a static method cannot access class attributes. It can access only the values passed in its parameter.

# (OOP) Concept in Python (cont...)



- Doesn't need self parameter
- Doesn't need self or cls as parameter
- Need decorator @staticmethod
- Can only access variables passed as argument.
- Static method cannot be accessed through class or it's instance.

No need of object (instance of class) to access the static method.
Can be access directly with classname.

@staticmethod decorator required

Cannot access class attributes. Only parameters passed to the method is accessible inside the method.

```
class Car:
    def __init__(self,make,model,year):
        self.make=make
        self.model=model
        self.year=year

    def details(self):
        print("Car Details ")
        print ('Make ',self.make)
        print('Model ',self.model
        print('Year ',self.year)

    @staticmethod
    def affordability(make):
        if make=='Toyota':
            return 'Affordoable'
        elif make=='Audi':
            return 'Luxury'
        else:
            return 'Unknown'
```

```
In [2]: Car.affordability('Audi')

Out[2]: 'Luxury'

In [3]: toyotaCamry=Car('Toyota','Camry','2007')

In [4]: toyotaCamry.details()

        Car Details
        Make  Toyota
        Model  Camry
        Year  2007

In [5]: toyotaCamry.affordability('Toyota')

Out[5]: 'Affordoable'
```

Can also access through object

static method is used to utlity method
class method is used state to the object

# (OOP) Concept in Python (cont...)

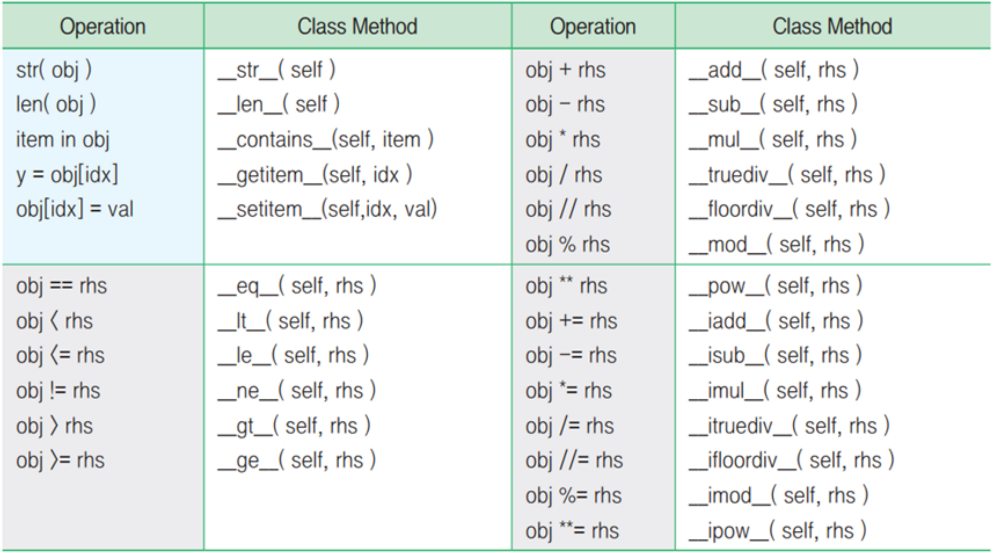
## Private/Protected Variables and Methods?

- Python does not support access control. In other words, all attributes are "public" and are accessible by ALL. There is no "private" attributes like C++/Java.
- However, by convention:
  - Names begin with an underscore (_) are meant for internal use, and are not recommended to be accessed outside the class definition.
  - Names begin with double underscores (_ _) and not end with double underscores are further hidden from direct access through name mangling (or rename).
  - Names begin and end with double underscores (such as __init__, __str__, __add__) are special magic methods

## Operator Overloading

- Python supports operator overloading
- You can overload '+', '-', '*', '/', '//' and '%' by overriding member methods __add__(), __sub__(), __mul__(), __truediv__(), __floordiv__() and __mod__(), respectively. You can overload other operators too

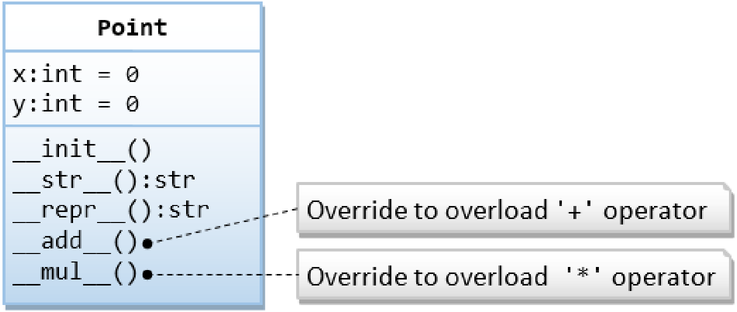
# (OOP) Concept in Python (cont...)



| Operation | Class Method | Operation | Class Method |
|---|---|---|---|
| str( obj ) | __str__( self ) | obj + rhs | __add__( self, rhs ) |
| len( obj ) | __len__( self ) | obj − rhs | __sub__( self, rhs ) |
| item in obj | __contains__(self, item ) | obj * rhs | __mul__( self, rhs ) |
| y = obj[idx] | __getitem__(self, idx ) | obj / rhs | __truediv__( self, rhs ) |
| obj[idx] = val | __setitem__(self,idx, val) | obj // rhs | __floordiv__( self, rhs ) |
| | | obj % rhs | __mod__( self, rhs ) |
| obj == rhs | __eq__( self, rhs ) | obj ** rhs | __pow__( self, rhs ) |
| obj 〈 rhs | __lt__( self, rhs ) | obj += rhs | __iadd__( self, rhs ) |
| obj 〈= rhs | __le__( self, rhs ) | obj −= rhs | __isub__( self, rhs ) |
| obj != rhs | __ne__( self, rhs ) | obj *= rhs | __imul__( self, rhs ) |
| obj 〉 rhs | __gt__( self, rhs ) | obj /= rhs | __itruediv__( self, rhs ) |
| obj 〉= rhs | __ge__( self, rhs ) | obj //= rhs | __ifloordiv__( self, rhs ) |
| | | obj %= rhs | __imod__( self, rhs ) |
| | | obj **= rhs | __ipow__( self, rhs ) |

# Point class

- Example: a Point class, which models a 2D point with x and y coordinates.
- the operators '+' and '*' are overloaded by overriding the so-called magic methods __add__() and __mul__().



**Point**

x:int = 0
y:int = 0

__init__()
__str__():str
__repr__():str
__add__()
__mul__()

Override to overload '+' operator

Override to overload '*' operator

- x and y variables are public and are accessible outside the class through the object instant
- Python Code

# Point class (cont...)

```python
class Point:
"""A Point instance models a 2D point with x and y coordinates"""

def __init__(self, x = 0, y = 0): # Initializer, it creates the instance variables x and y with default of (0, 0)
self.x = x
self.y = y

def __str__(self):   #Return a descriptive string for this instance
return '({}, {})'.format(self.x, self.y)

def __repr__(self):  # Return a command string to re-create this instance
return 'Point(x={}, y={})'.format(self.x, self.y)

def __add__(self, right): # Override the '+' operator:  create and return a new instance
p = Point(self.x + right.x, self.y + right.y)
return p

def __mul__(self, factor): # Override the '*' operator:  modify and return this instance
self.x *= factor
self.y *= factor
return self
```

## The getattr(), setattr(), hasattr() and delattr() Built-in Functions

- You can access an object's attribute via the dot operator by hard-coding the attribute name, provided you know the attribute name in compile time.
- For example, you can use
  - obj_name.attr_name: to read an attribute
  - obj_name.attr_name = value: to write value to an attribute
  - del obj_name.attr_name: to delete an attribute

# Point class (cont...)

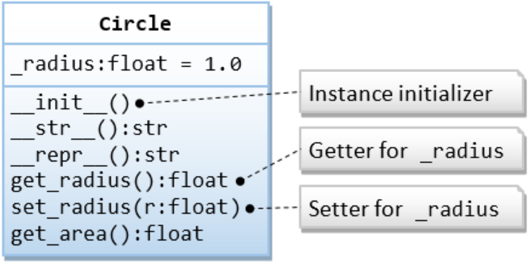
- Alternatively, you can use built-in functions like getattr(), setattr(), delattr(), hasattr(), by using a variable to hold an attribute name, which will be bound during runtime.
  - **hasattr(obj_name, attr_name) -> bool:** returns True if the obj_name contains the atr_name.
  - **getattr(obj_name, attr_name[, default]) -> value:** returns the value of the attr_name of the obj_name, equivalent to obj_name.attr_name. If the attr_name does not exist, it returns the default if present; otherwise, it raises AttributeError.
  - **setattr(obj_name, attr_name, attr_value):** sets a value to the attribute, equivalent to obj_name.attr_name = value.
  - **delattr(obj_name, attr_name):** deletes the named attribute, equivalent to del obj_name.attr_name.

```
class MyClass:
"""This class contains an instance variable called myvar"""

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

myinstance = MyClass(8)
print(myinstance.myvar)              # 8
print(getattr(myinstance, 'myvar'))  # 8
print(getattr(myinstance, 'no_var', 'default'))  # default
attr_name = 'myvar'
print(getattr(myinstance, attr_name))  # Using a variable
setattr(myinstance, 'myvar', 9)  # Same as myinstance.myvar = 9
print(getattr(myinstance, 'myvar'))  # 9
print(hasattr(myinstance, 'myvar'))  # True
delattr(myinstance, 'myvar')
print(hasattr(myinstance, 'myvar'))  # False
```

# Circle class

- The Circle class shall contain a data attribute radius and a method get_area(), as shown in the following class diagram.
- Python code

```
from math import pi
class Circle:
"""A Circle instance models a circle with a radius"""
def __init__(self, _radius = 1.0): #Initializer with default radius of 1.0
self.set_radius(_radius)   # Call setter
def set_radius(self, _radius): #Setter for instance variable radius
self._radius = _radius
def get_radius(self):   # Getter for instance variable radius
return self._radius
def get_area(self): # Return the area of this Circle instance
return self.get_radius() * self.get_radius() * pi  # Call getter
def __repr__(self): # Return a command string to recreate this instance
return 'Circle(radius={})'.format(self.get_radius())  # Call getter
```



Circle

_radius:float = 1.0

__init__()

__str__():str

__repr__():str

get_radius():float

set_radius(r:float)

get_area():float

Instance initializer

Getter for _radius

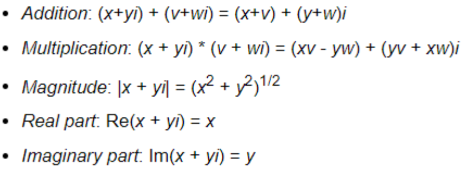
Setter for _radius

# Complex Numbers class

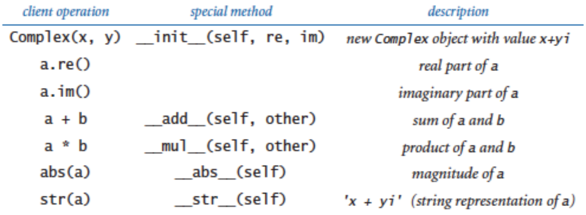
## Complex Numbers

- A complex number is a number of the form x + yi, where x and y are real numbers and i is the square root of -1. The number x is known as the real part of the complex number, and the number y is known as the imaginary part.

- The operations on complex numbers that are needed for basic computations



- *Addition*: $(x+yi) + (v+wi) = (x+v) + (y+w)i$
- *Multiplication*: $(x + yi) * (v + wi) = (xv - yw) + (yv + xw)i$
- *Magnitude*: $|x + yi| = (x^2 + y^2)^{1/2}$
- *Real part*: $\mathrm{Re}(x + yi) = x$
- *Imaginary part*: $\mathrm{Im}(x + yi) = y$



| *client operation* | *special method* | *description* |
|---|---|---|
| `Complex(x, y)` | `__init__(self, re, im)` | *new* `Complex` *object with value* `x+yi` |
| `a.re()` | | *real part of* `a` |
| `a.im()` | | *imaginary part of* `a` |
| `a + b` | `__add__(self, other)` | *sum of* `a` *and* `b` |
| `a * b` | `__mul__(self, other)` | *product of* `a` *and* `b` |
| `abs(a)` | `__abs__(self)` | *magnitude of* `a` |
| `str(a)` | `__str__(self)` | `'x + yi'` *(string representation of* `a`*)* |

*API for a user-defined* `Complex` *data type*

# The Bag Abstract Data Type using List

## The Bag Abstract Data Type

- A bag is a container that stores a collection in which duplicate values are allowed. The items, each of which is individually stored, have no particular order but they must be comparable.

| Operation | Description |
|---|---|
| Bag() | Creates a bag that is initially empty. |
| length() | Returns the number of items stored in the bag. Accessed using the len() function |
| contains(item) | contains ( item ): Determines if the given target item is stored in the bag and returns the appropriate Boolean value. Accessed using the *in* operator. |
| add(item) | Adds the given item to the bag. |
| remove(item) | Removes and returns an occurrence of item from the bag. An exception is raised if the element is not in the bag. |
| iterator() | Creates and returns an iterator that can be used to iterate over the collection of items. |

```
class Bag(object):
# Constructs an empty bag.
def __init__(self):
self.bag = []

# Returns the number of items in the bag.
def __len__(self):
return len(self.bag)
def __str__(self):
st=''
for item in self.bag:
st += str(item) + ' '
```

# The Bag Abstract Data Type using List (cont...)

```
return st;

# Determines if an item is contained in the bag.
def __contains__(self, item):
return item in self.bag

# Adds a new item to the bag.
def add(self, item):
self.bag.append(item)

# Removes and returns an instance of the item from the bag.
def remove(self, item):
assert item in self.bag, "The item must be in the bag"
ndx = self.bag.index(item)
return self.bag.pop(ndx)

# Returns an iterator for traversing the list of items.
def __iter__(self):
return _BagIterator(self.bag)
^^I
# An iterator for the Bag ADT implemented as a Python list.
class _BagIterator :
def __init__( self, theList=None ):
self._bagItems = theList
self._curItem = 0

def __iter__( self ):
return self

def __next__( self ):
if self._curItem < len( self._bagItems ) :
item = self._bagItems[ self._curItem ]
self._curItem += 1
return item
else :
raise StopIteration
```

# The Bag Abstract Data Type using List (cont...)