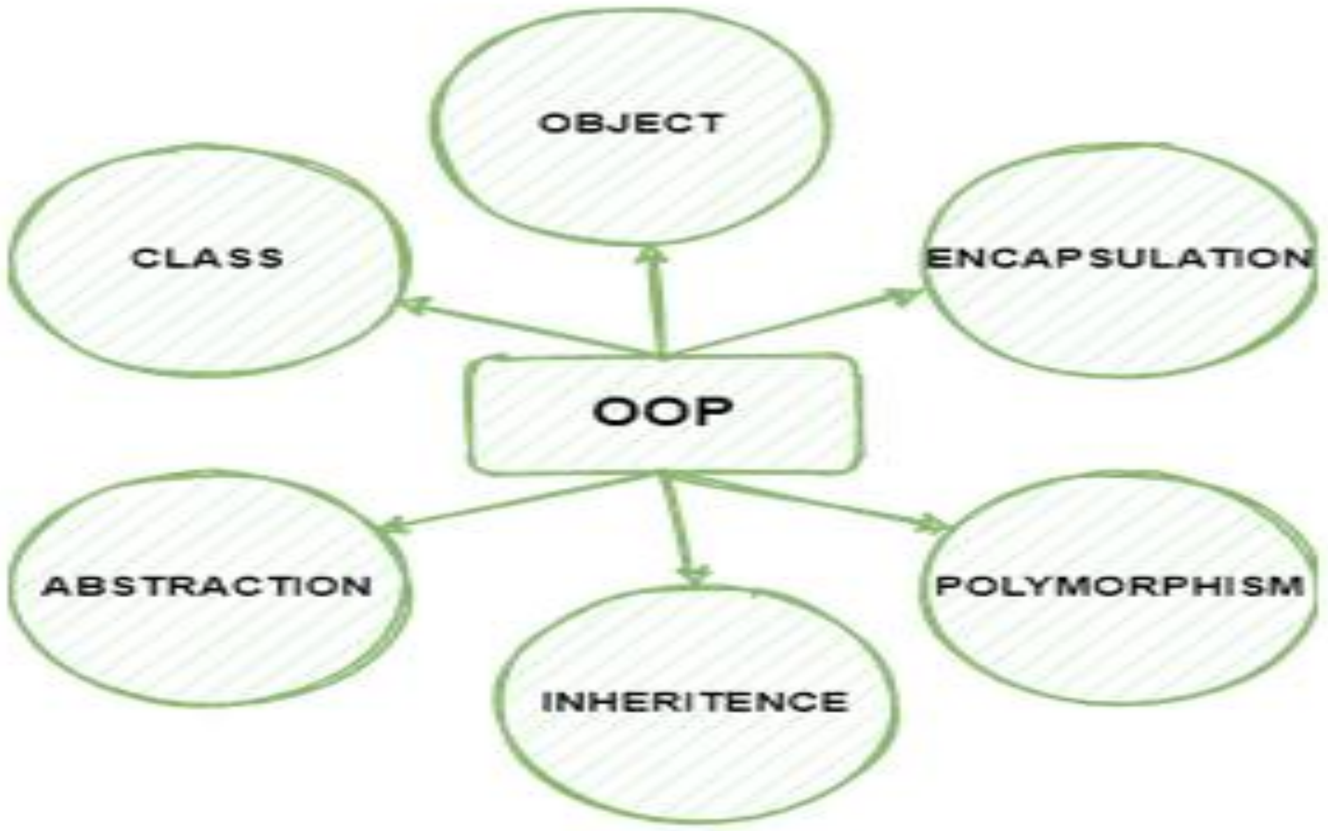
Classes and Objects

Objected oriented programming using Python



OOPs

Class in OOPS

- A programmer-defined type is also called a class.
- The class can be defined as a collection of objects.
  - class Point:
  - """Represents a point in 2-D space."""“
  - print(point)
- The header indicates that the new class is called Point. The body is a docstring that explains what the class is for.

Contd…

- Defining a class named Point creates a **class object**:
  - >>> Point
  - <class '__main__.Point'>
- The class object is like a factory for creating objects.
  - >>> blank = Point()
  - >>> blank
  - Print(blank)
  - <__main__.Point object at 0xb7e9d3ac>
- The return value is a reference to a Point object, which we assign to blank.

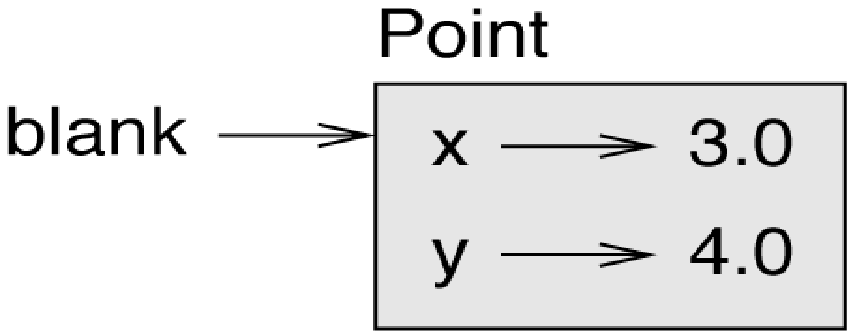
Objects

- Creating a new object is called **instantiation**, and the object is an **instance** of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).
- Every object is an instance of some class, so “object” and “instance” are interchangeable.

Attributes

- You can assign values to an instance using dot notation:
  - >>> blank.x = 3.0
  - >>> blank.y = 4.0
- In this case, we are assigning values to named elements of an object. These elements are called attributes.
- Attributes represent the characteristics of a class. When an object is instantiated and the values are assigned to attributes, they are then referred to as instance variables.

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```



Contd…

- You can use dot notation as part of any expression.
  - >>> '(%g, %g)' % (blank.x, blank.y)
  - '(3.0, 4.0)'
  - >>> distance = math.sqrt(blank.x**2 + blank.y**2)
  - >>> distance
  - 5.0
- You can pass an instance as an argument in the usual way.
  - def print_point(p):
  - print('(%g, %g)' % (p.x, p.y))
- print_point takes a point as an argument and displays it in mathematical notation.
  - >>> print_point(blank)
  - (3.0, 4.0)
- Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

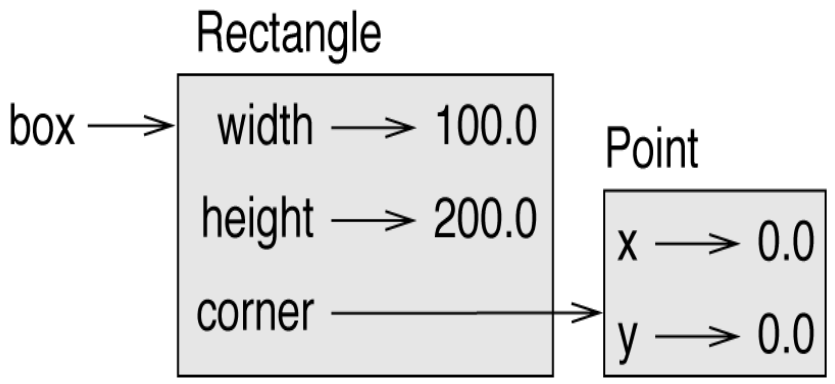
Rectangles

- There are at least two possibilities:
  - You could specify one corner of the rectangle (or the center), the width and the height.
  - You could specify two opposing corners.

  Class definition:
  - class Rectangle:
  - """Represents a rectangle.
  - attributes: width, height, corner.
  - ""“
- The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

Contd…

- To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes
  - box = Rectangle()
  - box.width = 100.0
  - box.height = 200.0
  - box.corner = Point()
  - box.corner.x = 0.0
  - box.corner.y = 0.0



Instances as Return Values

- Functions can return instances.
  - def find_center(rect):
  - p = Point()
  - p.x = rect.corner.x + rect.width/2
  - p.y = rect.corner.y + rect.height/2
  - return p
- passes box as an argument and assigns the resulting Point to center:
  - >>> center = find_center(box)
  - >>> print_point(center)
  - (50, 100)

Objects Are Mutable

- You can change the state of an object by making an assignment to one of its attributes.
  - box.width = box.width + 50
  - box.height = box.height + 100
- You can also write functions that modify objects.
  - def grow_rectangle(rect, dwidth, dheight):
  - rect.width += dwidth
  - rect.height += dheight
- Here is an example that demonstrates the effect:
  - >>> box.width, box.height
  - (150.0, 300.0)
  - >>> grow_rectangle(box, 50, 100)
  - >>> box.width, box.height
  - (200.0, 400.0)

Copying

- Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

Contd…

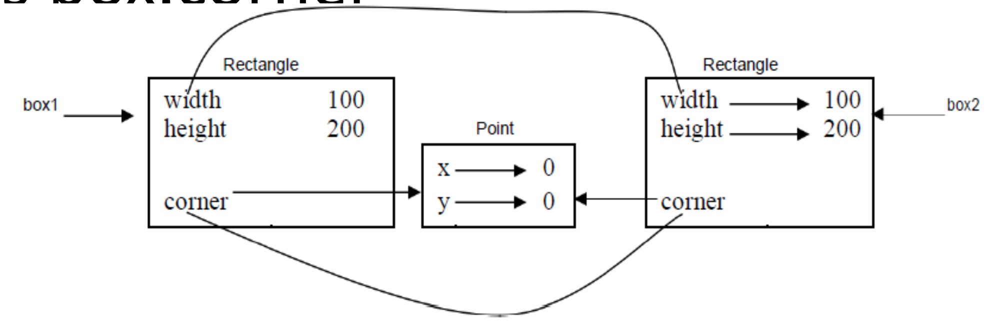
- >>> p1 = Point()
- >>> p1.x = 3.0
- >>> p1.y = 4.0
- >>> import copy
- >>> p2 = copy.copy(p1)

• p1 and p2 contain the same data, but they are not the same Point:

- >>> print_point(p1)
- (3, 4)
- >>> print_point(p2)
- (3, 4)
- >>> p1 is p2
- False
- >>> p1 == p2
- False

copy.copy

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point:
  - >>> box2 = copy.copy(box)
  - >>> box2 is box
  - False
  - >>> box2.corner is box.corner
  - True



deep copy

- deepcopy that copies not only the object but also the objects it refers to and the objects they refer to and so on.
  - >>> box3 = copy.deepcopy(box)
  - >>> box3 is box
  - False
  - >>> box3.corner is box.corner
  - False
  - box3 and box are completely separate objects.

Debugging

- If you try to access an attribute that doesn’t exist, you get an AttributeError:
  - >>> p = Point()
  - >>> p.x = 3
  - >>> p.y = 4
  - >>> p.z
- AttributeError: Point instance has no attribute 'z'

Contd…

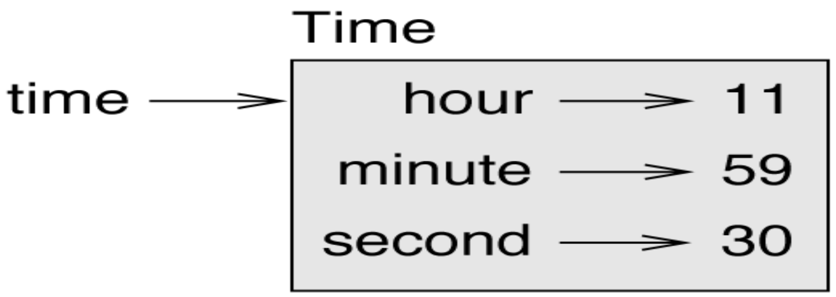
- You can also use a try statement to see if the object has the attributes you need:
  - try:
  - x = p.x
  - except AttributeError:
  - x = 0
- If you are not sure what type an object is, you can ask:
  - >>> type(p)
  - <class '__main__.Point'>
- You can also use isinstance to check whether an object is an instance of a class:
  - >>> isinstance(p, Point)
  - True

Contd…

- If you are not sure whether an object has a particular attribute, you can use the builtin function hasattr:
  - >>> hasattr(p, 'x')
  - True
  - >>> hasattr(p, 'z')
  - False
- The first argument can be any object; the second argument is a string that contains the name of the attribute.

Classes and functions

- Time
- Define a class called Time that records the time of day.
- class definition
  - class Time:
  - """Represents the time of day.
  - attributes: hour, minute, second
  - """
- We can create a new Time object and assign attributes for hours, minutes and seconds:
  - time = Time()
  - time.hour = 11
  - time.minute = 59
  - time.second = 30



Pure functions

```
class Time:
    """Represents the time of a day
    Attributes: hour, minute, second """
def printTime(t):
    print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))
def add_time(t1,t2):
    sum=Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
```

Contd…

```
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum

t1=Time()

t1.hour=10
t1.minute=34
t1.second=25
print("Time1 is:")
printTime(t1)

t2=Time()

t2.hour=2
t2.minute=12
t2.second=41
print("Time2 is :")
printTime(t2)

t3=add_time(t1,t2)

print("After adding two time objects:")
printTime(t3)
```

Pure functions

- The function add_ time() takes two arguments of type Time, and returns a Time object, whereas, it is not modifying contents of its arguments t1 and t2. Such functions are called as pure functions.
- The function creates a new Time object, initializes its attributes and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

Modifiers

- Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.
- increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier.

Contd…

```
def increment(t, seconds):
    t.second += seconds
    while t.second >= 60:
        t.second -= 60
        t.minute += 1
    while t.minute >= 60:
        t.minute -= 60
        t.hour += 1
```

Classes and Methods

- Object-Oriented Features
- Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining characteristics:
  - Programs include class and method definitions.
  - Most of the computation is expressed in terms of operations on objects.
  - Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

Methods

- A method is a function that is associated with a particular class.
  - Methods are semantically the same as functions, but there are two syntactic differences:
  - Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
  - The syntax for invoking a method is different from the syntax for calling a function.

Printing Objects

  - class Time:
  - """Represents the time of day."""
  - def print_time(time):
  - print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
- To call this function, you have to pass a Time object as an argument:
  - >>> start = Time()
  - >>> start.hour = 9
  - >>> start.minute = 45
  - >>> start.second = 00
  - >>> print_time(start)
- 09:45:00

Contd…

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

- >>> Time.print_time(start)
- 09:45:00
- >>> start.print_time()
- 09:45:00

- In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject.**
- Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
- By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The init Method

- The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is __init__ (two underscore characters, followed by init, and then two more underscores).
  - def __init__(self, hour=0, minute=0, second=0):
  - self.hour = hour
  - self.minute = minute
  - self.second = second

Contd…

- It is common for the parameters of __init__ to have the same names as the attributes. The statement
  - self.hour = hour
- stores the value of the parameter hour as an attribute of self.
- The parameters are optional, so if you call Time with no arguments, you get the default values:
  - >>> time = Time()
  - >>> time.print_time()
  - 00:00:00
- If you provide one argument, it overrides hour:
  - >>> time = Time (9)
  - >>> time.print_time()
  - 09:00:00
- If you provide two arguments, they override hour and minute:
  - >>> time = Time(9, 45)
  - >>> time.print_time()
  - 09:45:00
- And if you provide three arguments, they override all three default values.

The __str__ Method

- __str__ is a special method, like __init__, that is supposed to return a string representation of an object.
  - def __str__(self):
  - return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
- When you print an object, Python invokes the str method:
  - >>> time = Time(9, 45)
  - >>> print(time)
  - 09:45:00
- When you write a new class, start by writing __init__, which makes it easier to instantiate objects and __str__, which is useful for debugging.

Operator overloading

- Ability of an existing operator to work on user-defined data type (class) is known as operator overloading.

- **Operator Overloading** means giving extended meaning beyond their predefined operational meaning.

For example operator + is used

- To add two integers
- To join two strings
- To merge two lists.

- It is achievable because '+' operator is overloaded by int class and str class.
- The same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

Contd…

Operator	Special Function in Python	Operator	Special Function in Python
+	__add__()	<=	__le__()
-	__sub__()	>=	__ge__()
*	__mul__()	==	__eq__()
/	__truediv__()	!=	__ne__()
%	__mod__()	in	__contains__()
<	__lt__()	len	__len__()
>	__gt__()	str	__str__()

Operator Overloading

- By defining other special methods, you can specify the behavior of operators on programmer-defined types
  - def __add__(self, other):
  - seconds = self.time_to_int() + other.time_to_int()
  - return int_to_time(seconds)
  - >>> start = Time(9, 45)
  - >>> duration = Time(1, 35)
  - >>> print(start + duration)
  - 11:20:00
- When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__.

Type-Based Dispatch

```
– def __add__(self, other):
        if  isinstance(other, Time):
            return self.add_time(other)
      else:
            return self.increment(other)
– def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
          return int_to_time(seconds)
– def increment(self, seconds):
        seconds += self.time_to_int()
          return int_to_time(seconds)
```

Contd…

- If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.
  - >>> start = Time(9, 45)
  - >>> duration = Time(1, 35)
  - >>> print(start + duration)
  - 11:20:00
  - >>> print(start + 1337)
  - 10:07:17
  - >>> print(1337 + start)
  - TypeError: unsupported operand type(s) for +: 'int' and 'instance'

Contd…

- The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn’t know how. But there is a clever solution for this problem: the special method __radd__, which stands for “right-side add”.
- This method is invoked when a Time object appears on the right side of the + operator.
  - def __radd__(self, other):
  - return self.__add__(other)
  - >>> print(1337 + start)
  - 10:07:17

Polymorphism

- The word polymorphism means having many forms.
- In programming, polymorphism means the same function name (but different signatures) being used for different types.
- The key difference is the data types and number of arguments used in function

Contd…

- Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Contd…

– def histogram(s):
– d = dict()
– for c in s:
– if c not in d:
– d[c] = 1
– else:
– d[c] = d[c]+1
– return d

Contd…

- >>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
- >>> histogram(t)
- {'bacon': 1, 'egg': 1, 'spam': 4}

* Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse.
  - >>> t1 = Time(7, 43)
  - >>> t2 = Time(7, 41)
  - >>> t3 = Time(7, 37)
  - >>> total = sum([t1, t2, t3])
  - >>> print(total)
  - 23:01:00

Example :Student class in Python

```
class Student:
     def __init__(self,USN,SName):
        self.USN = USN
        self.Sname = SName

    def StudDisp(self):
        print('\n Student Details')
        print('Student USN: ',self.USN )
        print('Student Name:',self.Sname )
```

Student example

```
class Student:
    def __init__(self,USN,SName, phy, che, maths):
                self.USN = USN
               self.Sname = SName
               self.phy = phy
                self.che = che
                self.maths = maths
                self.total = 0
     # compute total method to initalize the student data members
    def compute_tot(self):
                self.total = self.phy+self.che+self.maths

#Stud_disp() method to display the students details
    def Stud_Disp(self):
               print('\n Student Details')
               print('Student USN: ',self.USN )
               print('Student Name:',self.Sname )
               print('Total score:',self.compute_tot())
```

Continued …

```
def main():
   # create instance of Student class S1 and calls __init__()
   S1= Student('1SI18MCA25',‘ Asha Gowda',44,45,46)


   # display contents of Student instance S1
   S1.Stud_Disp()

   # create instance of Student class S2 and calls __init__()

   S2 = Student('1SI18CS125','Bhargavi', 20,30,40)

   # display contents of Student instance S2
   S2.Stud_Disp()


main()
```

Adding two complex numbers

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def add_complex(self, c1,c2):
        self.real = c1.real + c2.real
        self.imag = c2.imag + c2.imag

        print(f"{self.real}+{self.imag}i")

   ==========================================
c1 = Complex(10,5)
c2 = Complex(2,4)
c3 = Complex(0,0)
c3.add_complex(c1,c2)
```

Over loading ‘+’ for adding two complex numbers and __str__()

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real+ other.real,  self.imag + other.imag)

    def __str__(self):
        return str(f"{self.real}+{self.imag}i")

 c1 = Complex(10,5)
c2 = Complex(2,4)
c3 = c1+c2
print(c3)
```

# A simple Python function to demonstrate Polymorphism

```
def add(x, y, z = 0):
    return x + y + z

print(add(2, 3))
print(add(2, 3, 4))
```

Python program to demonstrate in-built poly-morphic functions

```
# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Exception handling Python coding

Common errors in Python coding

IndentationError	Raised when indentation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
NameError	Raised when a variable does not exist
ZeroDivisionError	Raised when the second operator in a division is zero

Examples of error

```
>>> L1 = [4,90, 'Tumkur',80.89]
>>> L1[5]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    L1[5]
IndexError: list index out of range

>>> a = 9
>>> b =0
>>> a/b
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a/b
ZeroDivisionError: division by zero
```

Continued…

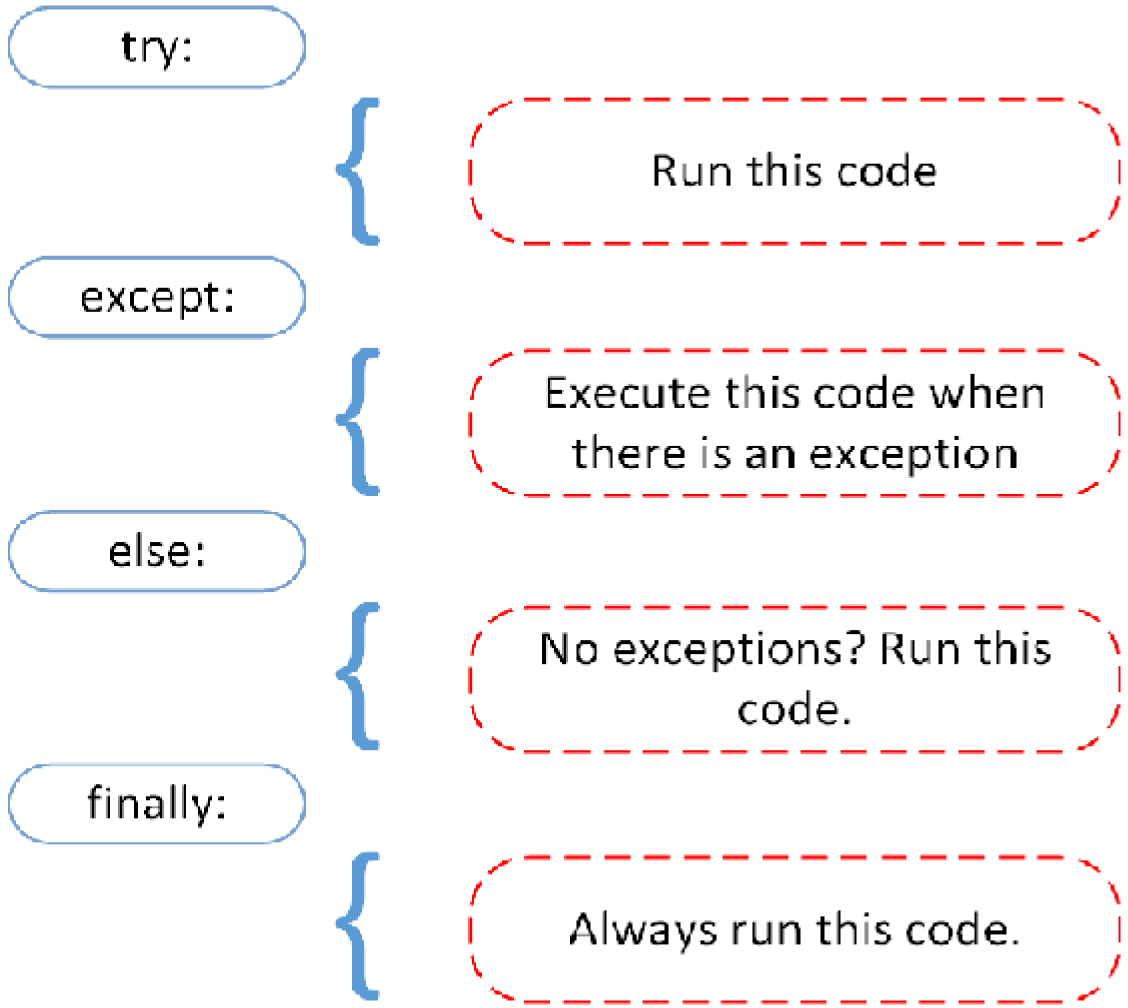
```
>>> L1 = [ 4,5,’abc’, 10]
>>> sum(L1)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    sum(L1)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
a = 5, b = 10
>>> z = a+b+c
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
z = a+b+c
NameError: name 'c' is not defined
```

Continued…

```
>>>import math
>>> math.sqrt(-100)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    math.sqrt(-100)
ValueError: math domain error
```

Error handling in Python


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Program to illustrate ZeroDivision error

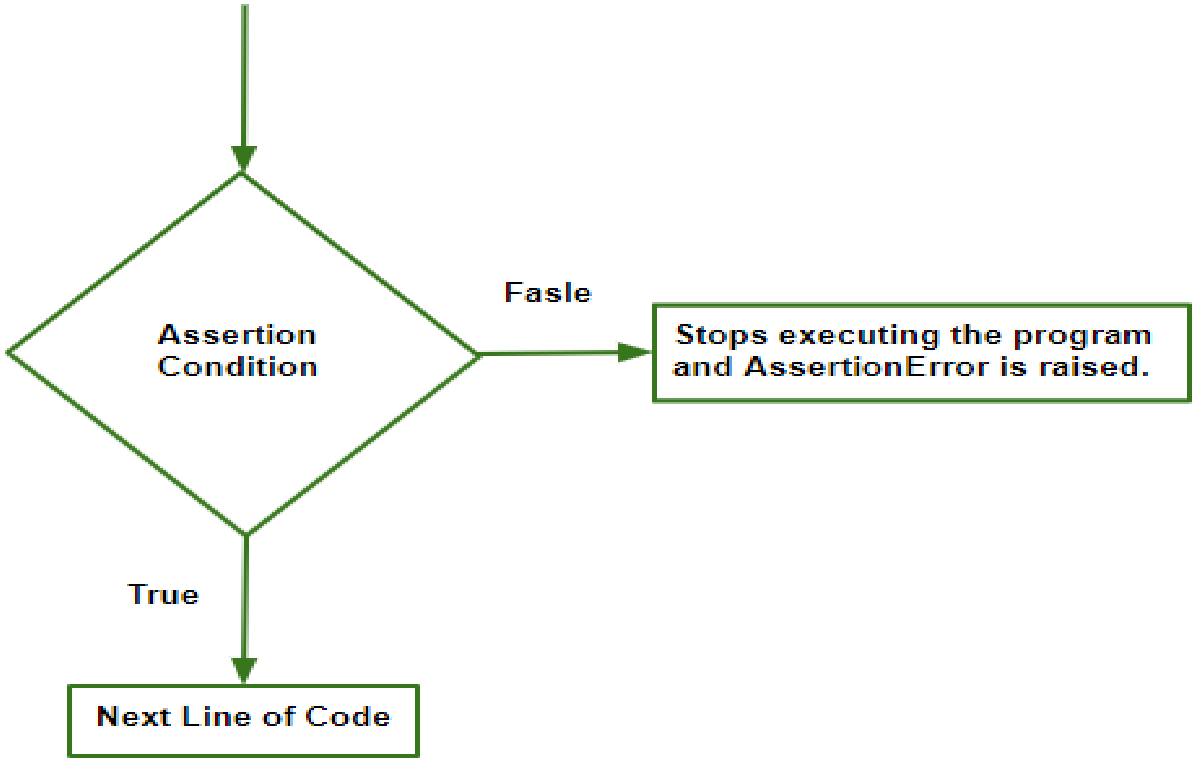
```
while(1):
   try:
      x=int(input('Enter a number: '))
      y=int(input('Enter another number: '))
      z=x/y

   except ZeroDivisionError:
            print("Division by 0 not accepted")
   else:
            print("Division = ", z)
            break
```

Assertion Error

If the condition is *True*, the control simply moves to the next line of code.

In case if it is *False* the program stops running and returns *AssertionError* Exception.



- **Syntax of assertion:**
  assert condition, error_message(optional)

```
# assert without try catch
x = 1
y = 0
assert y != 0, "Invalid Operation"
print(x / y)
```

Assert with try catch

```
try:
    x = 1
    y = 0
    assert y != 0, "Invalid Operation"
    print(x / y)

except AssertionError as msg:
    print(msg)
```

Alternate way

```
try:
    x = 1
    y = 0
    assert y != 0
    print(x / y)

except AssertionError as msg:
    print("Invalid Operation")
```