Python

Trainer: Mirza Touseef

Riphah Institute of System Engineering

Contents

Modules vs Packages

Python Modules

- A module is a file that contains Python code that we can use in our program.

- There are several built-in functions like print(), input() and sum() that are readily available to us. In addition to these functions, Python also has several functions that are defined inside a module which we can use after we import them.

- Let's use one such module called math

Modules

```python
import math

number = 25
result = math.sqrt(number)
print(result)

print(math.pi)
```

Renaming Modules

```
import math as m

number = 25
result = m.sqrt(number)
print(result)

print(m.pi)
```

Python from…import statement

```python
from math import pi, sin, sqrt

value = sin(pi/2)
print(value)

num = sqrt(64)
print(num)
```

Python from...import statement

```python
from math import *

value = sin(pi/2)
print(value)

num = sqrt(64)
print(num)
```

Python Packages

A package is a directory containing multiple modules and other sub-packages.

Suppose we are developing a game with multiple objects, so it may have these different modules.

- player.py
- boss.py
- gun.py
- knife.py

Since these modules are in the same location, they look cluttered. We can structure them in this way:

- game
    - characters
        - player.py
        - boss.py
    - weapons
        - gun.py
        - knife.py

Python Packages

- Create a directory named game that will contain all our game components.
- We also need to create an **__init__.py** file inside game directory. This special file tells Python that this directory is a Python package.
- Also create the characters package with an **__init__.py** file.
- Now, create player.py and boss.py modules inside characters package.

Cont.

```
import game.characters.boss
import game.weapons.gun

game.characters.boss.get_boss_info()
game.weapons.gun.get_gun_info()
•
```

Classes

instances,
attributes,
methods,
class and instance data.

Python Object Oriented Programming

- Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of **objects** and **classes**.

- An object is any entity that has **attributes** and **behaviours**. For example, a parrot is an object. It has **attributes**: name, age, color, etc. and **behaviour**: dancing, singing, etc.
- Similarly, a class is a blueprint for that object.

Python Classes

- A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

- Since many houses can be made from the same description, we can create many objects from a class.

Define Python Class

- We use the class keyword to create a class in Python. For example,

```
class ClassName: # class definition
```

Here, we have created a class named ClassName.

- Let's see an example,

```
class Bike:
    name = ""
    gear = 0
```

Here,

- Bike is the name of the class
- name/gear are variables inside the class with default values "" and **0** respectively.
- The variables inside a class are called attributes.

Python Objects

- An object is called an instance of a class. For example, suppose Bike is a class then we can create objects like bike1, bike2, etc from the class.

- Here's the syntax to create an object:

```
objectName = ClassName()
```

- Let's see an example,

```
# create class
class Bike:
        name = ""
        gear = 0
# create objects of class
bike1 = Bike()
```

Here, bike1 is the object of the class. Now, we can use this object to access the class attributes.

Cont.

Here, student1 and student2 are objects of the Student class.

```
class Student:
    pass

student1 = Student()
student2 = Student()
```

Access Class Attributes Using Objects

- We use the . notation to access the attributes of a class. For example,

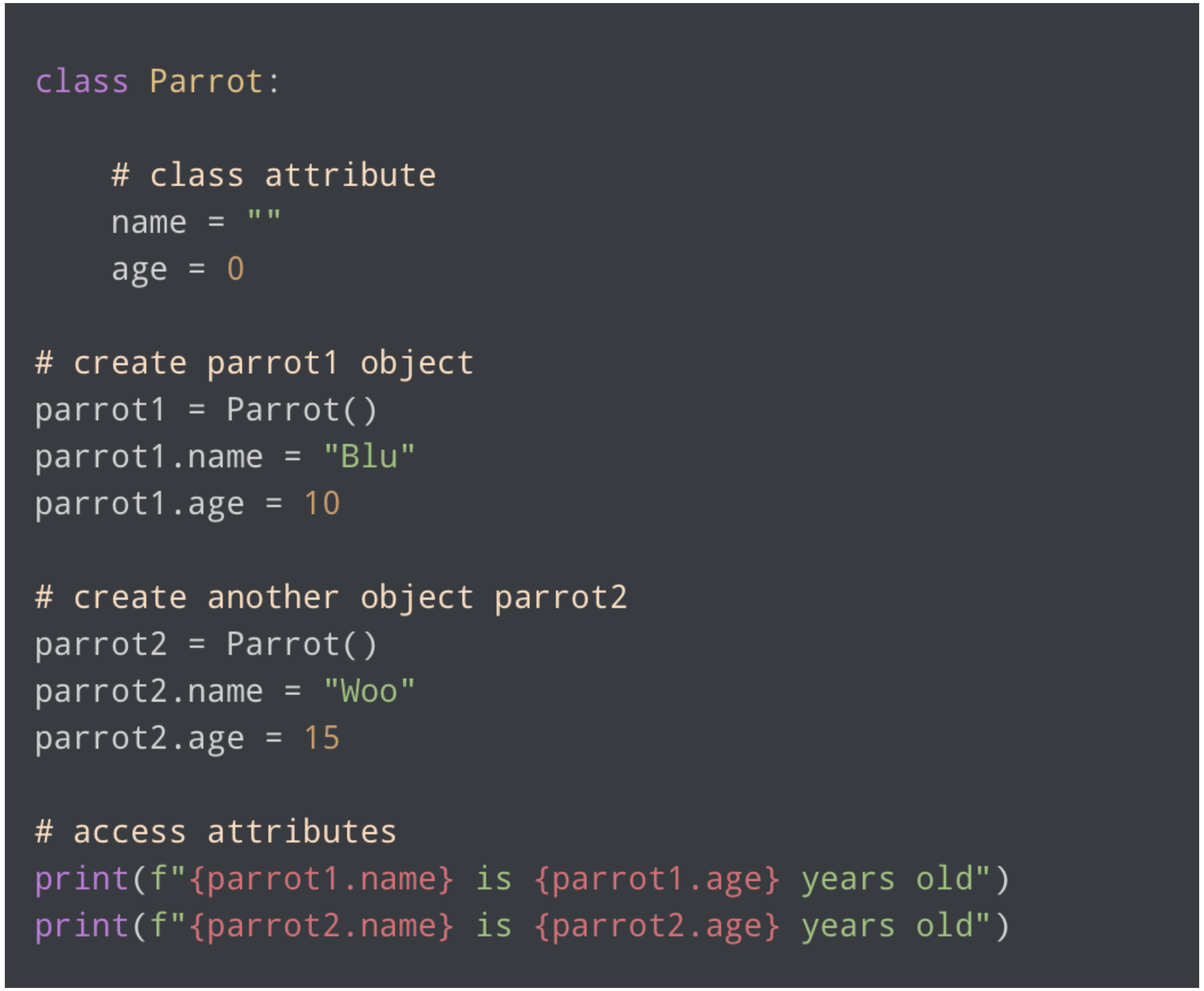
```
# modify the name attribute
bike1.name = "Mountain Bike”
 # access the gear attribute
bike1.gear
```

Here, we have used bike1.name and bike1.gear to change and access the value of name and gear attribute respectively.

Cont.

```
class Student:
    pass

student1 = Student()
student1.name = "Harry"
student1.marks = 85

print(student1.name)
print(student1.marks)
```

Python Class and Object



```python
class Parrot:

    # class attribute
    name = ""
    age = 0

# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10

# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15

# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```

Python Class and Object

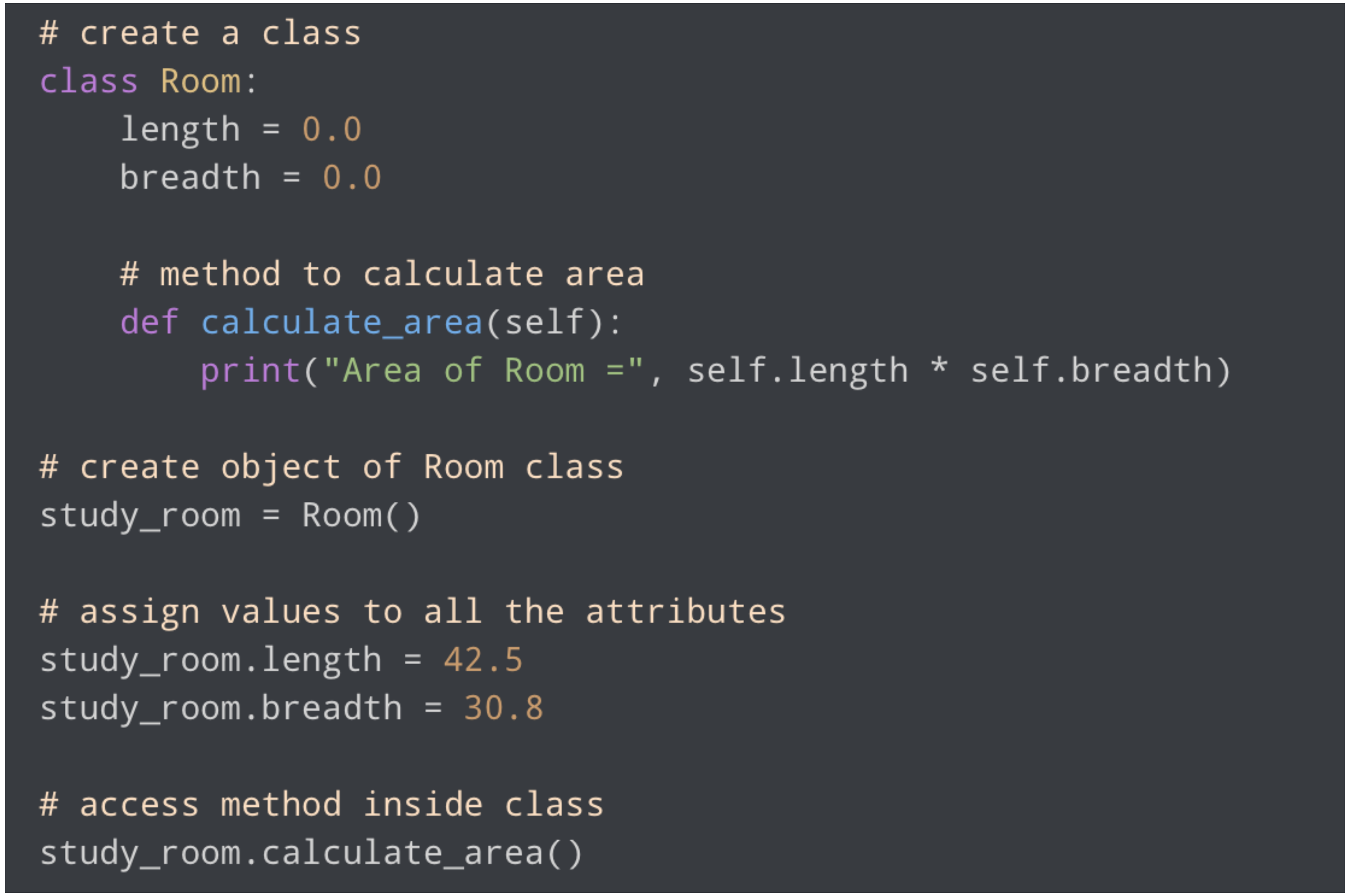
Output:

```
Blu is 10 years old
Woo is 15 years old
```

- In the above example, we created a class with the name Parrot which has two attributes: name and age.
- Then, we created instances of the Parrot class. Here, parrot1 and parrot2 are references (value) to our new objects.
- We then accessed and assigned different values to the instance attributes using the objects name and the . notation.

Python Methods

- We can also define a function inside a Python class. A Python function defined inside a class is called a method.



```python
# create a class
class Room:
    length = 0.0
    breadth = 0.0

    # method to calculate area
    def calculate_area(self):
        print("Area of Room =", self.length * self.breadth)

# create object of Room class
study_room = Room()

# assign values to all the attributes
study_room.length = 42.5
study_room.breadth = 30.8

# access method inside class
study_room.calculate_area()
```

Cont.

```
class Student:
    def check_pass_fail(self):
        if self.marks >= 40:
            return True
        else:
            return False

student1 = Student()
student1.name = "Harry"
student1.marks = 85

did_pass = student1.check_pass_fail()
print(did_pass)
```

Python Methods

Output:

```
Area of Room = 1309.0
```

In the above example, we have created a class named Room with:

- **Attributes**: length and breadth
- **Method**: calculate_area()

Here, we have created an object named study_room from the Room class. We then used the object to assign values to attributes: length and breadth.

Notice that we have also used the object to call the method inside the class,

Python Constructors

- Earlier we assigned a default value to a class attribute.

```
class Bike:
    name = ""
...
# create object
bike1 = Bike()
```

- However, we can also initialize values using the constructors. For example,

```
class Bike:

    # constructor function
    def __init__(self, name = ""):
        self.name = name

bike1 = Bike()
```

Python Constructors

- Here, __init__() is the constructor function that is called whenever a new object of that class is instantiated.

- The constructor above initializes the value of the name attribute. We have used the self.name to refer to the name attribute of the bike1 object.

- If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

```
bike1 = Bike("Mountain Bike")
```

- Here, "Mountain Bike" is passed to the name parameter of __init__().

Cont.

```python
class Student:

    def __init__(self, name, marks):
        self.name = name
        self.marks = marks


student1 = Student("Harry", 85)
print(student1.name)
print(student1.marks)
```

Activity - 1

Make a module which will have a class StudentData. The attributes of the class will be Name, RollNumber, Discipline and Gender. Keep on storing the data entered by the user for different students in separate objects from a method called EnterData(), until the user enters “done”. Lastly, print the student data on separate lines from a method called PrintData().

Activity 2

Create our own Complex class to add two complex numbers manually. If you do not know, a complex number has real and imaginary parts. When we add two complex numbers, we need to add real and imaginary parts separately.

```
n1 = Complex(5, 6)
n2 = Complex(-4, 2)
```

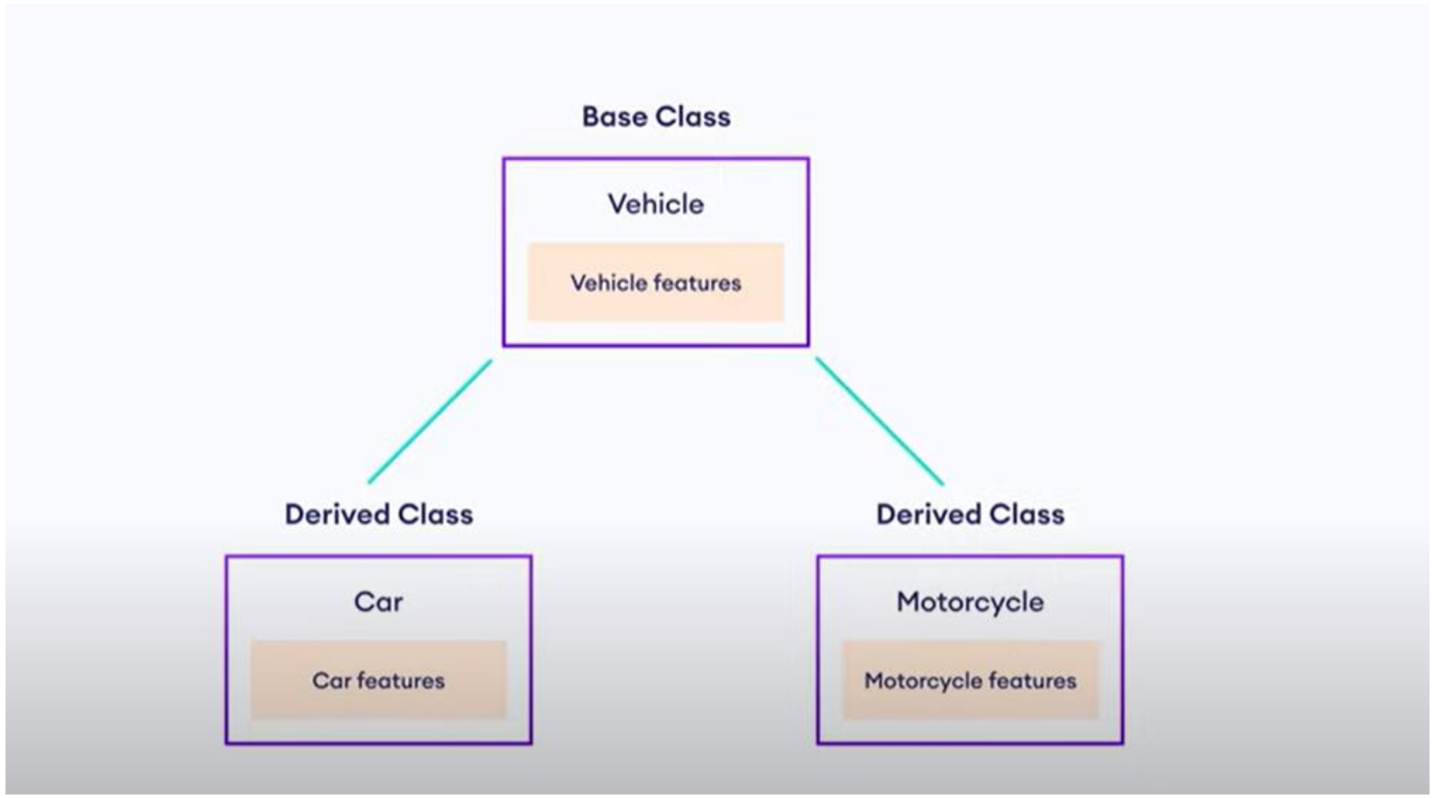
```python
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def add(self, number):
        real = self.real + number.real
        imag = self.imag + number.imag
        result = Complex(real, imag)
        return result

n1 = Complex(5, 6)
n2 = Complex(-4, 2)
result = n1.add(n2)

print("real =", result.real)
print("imag =", result.imag)
```

Python Inheritance

- Inheritance is a way of creating a new class for using details of an existing class without modifying it.

- The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example


Base Class
Vehicle
Vehicle features
Derived Class
Car
Car features
Derived Class
Motorcycle
Motorcycle features

Use of Inheritance in Python

```
class Animal:
    def eat():
        print("I can eat")

class Dog(Animal):
    def bark(self):
        print("I can bark")

class Cat(Animal):
    def get_grumpy(self):
        print("I am getting grumpy.")

dog1 = Dog()

dog1.bark()
dog1.eat()

cat1 = Cat()
cat1.eat()
```

Use of Inheritance in Python

Output:

```
I can bark
I can eat
I can eat
```

Here, dog1 (the object of derived class Dog) can access members of the base class Animal. It's because Dog is inherited from Animal.

```
dog1.bark()
dog1.eat()
```

Example of Inheritance

```python
class Polygon:
    def __init__(self, sides):
        self.sides = sides

    def display_info(self):
        print("A polygon is a two dimensional shape with straight lines")

    def get_perimeter(self):
        perimeter = sum(self.sides)
        return perimeter


class Triangle(Polygon):
    def display_info(self):
        print("A triangle is a polygon with 3 edges")
```

```python
class Polygon:
    def __init__(self, sides):
        self.sides = sides

    def display_info(self):
        print("A polygon is a two dimensional shape with straight
lines")

    def get_perimeter(self):
        perimeter = sum(self.sides)
        return perimeter


class Triangle(Polygon):
    def display_info(self):
        print("A triangle is a polygon with 3 edges")


class Quadrilateral(Polygon):
    def display_info(self):
        print("A quadrilateral is a polygon with 4 edges")
```

Method Overriding

If you have noticed, in the above example, we have the display_info() method in both our base class and the derived classes.

```
t1 = Triangle([5, 6, 7])
perimeter = t1.get_perimeter()
print("Perimeter:", perimeter)

t1.display_info()
```

Activity - 3

Make two classes Employee and HR. The attributes of Employee will be Name, Gender, CNIC and Cell. The attributes of HR will be Designation, Salary and Department. Inherit the Employee class into the HR class. Make a method of PrintData() in HR class and print the employee data in the following format through it:

Employee Name, Gender, CNIC, Cell

Designation, Salary, Department

```python
class Employee:
    def __init__(self, name, gender, cnic, cell):
        self.name = name
        self.gender = gender
        self.cnic = cnic
        self.cell = cell

class HR(Employee):
    def __init__(self, name, gender, cnic, cell, designation, salary, department):
        # Call the constructor of the base class (Employee)
        super().__init__(name, gender, cnic, cell)

        self.designation = designation
        self.salary = salary
        self.department = department

    def PrintData(self):
        employee_data = f"{self.name}, {self.gender}, {self.cnic}, {self.cell}"
        hr_data = f"{self.designation}, {self.salary}, {self.department}"

        print(employee_data)
        print(hr_data)

# Example usage:
employee = HR("John Doe", "Male", "12345-6789", "555-555-5555", "Manager", 50000, "HR")
employee.PrintData()
```

Encapsulation

- Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class.

- It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve **data hiding**.

- In Python, we denote private attributes using underscore as the prefix i.e single _ or double __. For example,

Encapsulation

```python
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Encapsulation

Output:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

- In the above program, we defined a class Computer.
- We used __init__() method to store the maximum selling price of Computer. Here, notice the code:

```
c.__maxprice = 1000
```

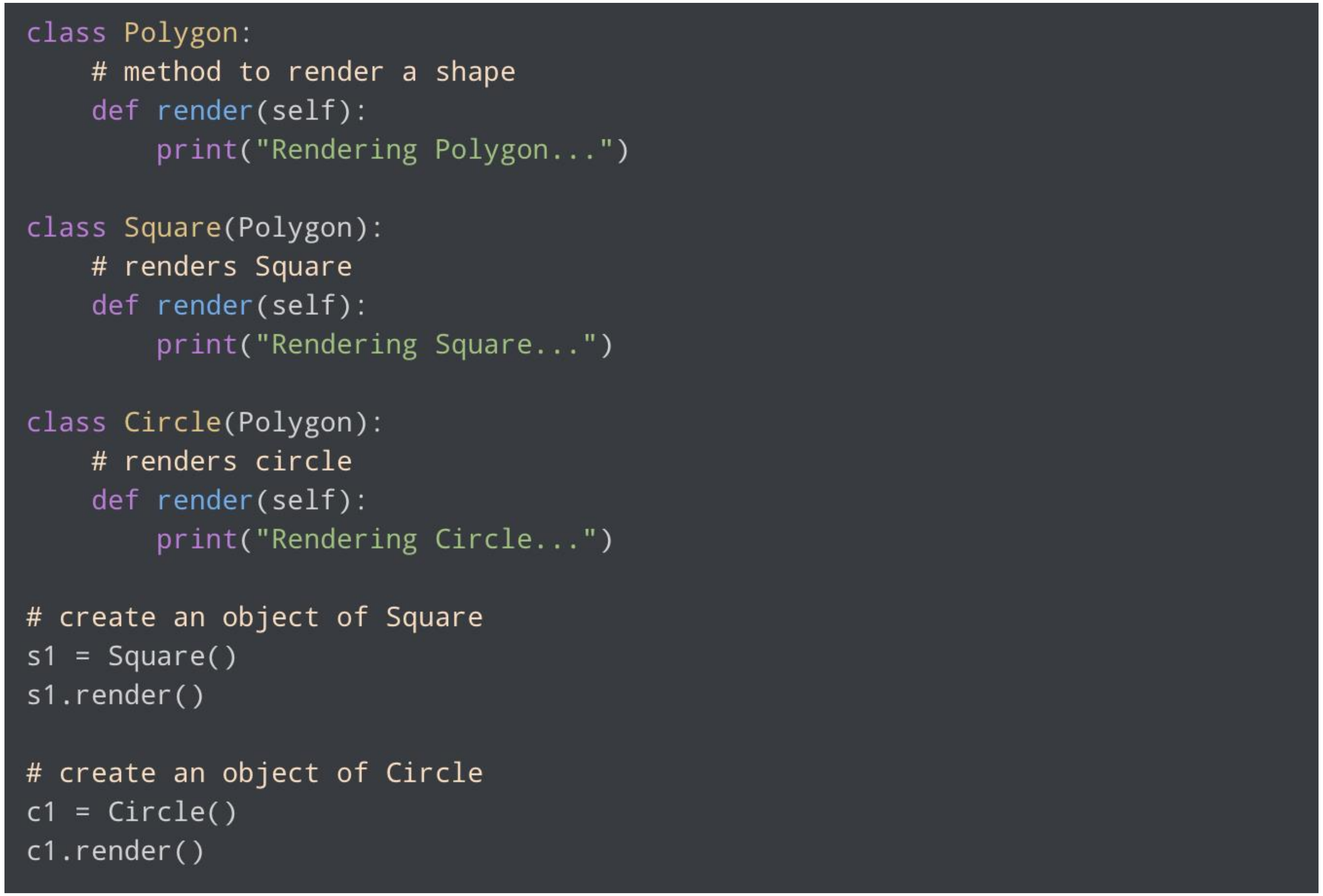
- Here, we have tried to modify the value of __maxprice outside of the class. However, since __maxprice is a private variable, this modification is not seen on the output.
- As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

Polymorphism

- Polymorphism is another important concept of object-oriented programming. It simply means more than one form.

- That is, the same entity (method or operator or object) can perform different operations in different scenarios.

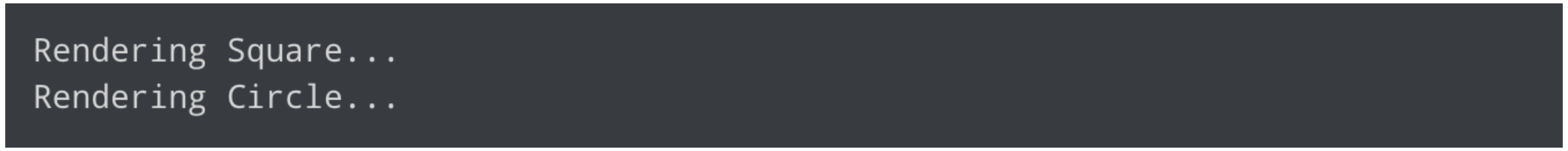
Polymorphism



```python
class Polygon:
    # method to render a shape
    def render(self):
        print("Rendering Polygon...")

class Square(Polygon):
    # renders Square
    def render(self):
        print("Rendering Square...")

class Circle(Polygon):
    # renders circle
    def render(self):
        print("Rendering Circle...")

# create an object of Square
s1 = Square()
s1.render()

# create an object of Circle
c1 = Circle()
c1.render()
```

Polymorphism

Output:



```
Rendering Square...
Rendering Circle...
```

- In the above example, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method.
- The main purpose of the render() method is to render the shape. However, the process of rendering a square is different from the process of rendering a circle.
- Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.

Operator Overloading

- In Python, we can change the way operators work for user-defined types.
- For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- This feature in Python that allows the same operator to have different meaning according to the context is called **operator overloading**.

Python Special Functions

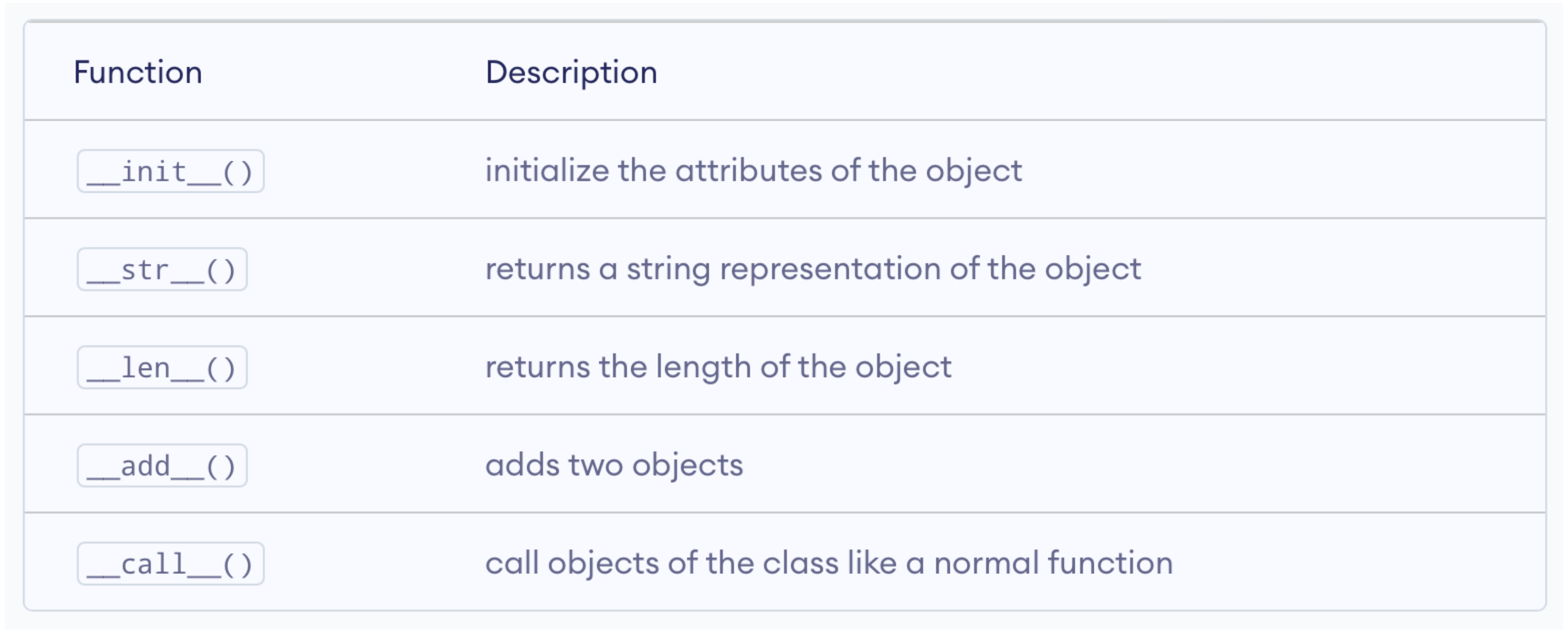
- Class functions that begin with double underscore __ are called special functions in Python.

- The special functions are defined by the Python interpreter and used to implement certain features or behaviours.

- They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as __init__() or __add__().

Python Special Functions

- Here are some of the special functions available in Python:



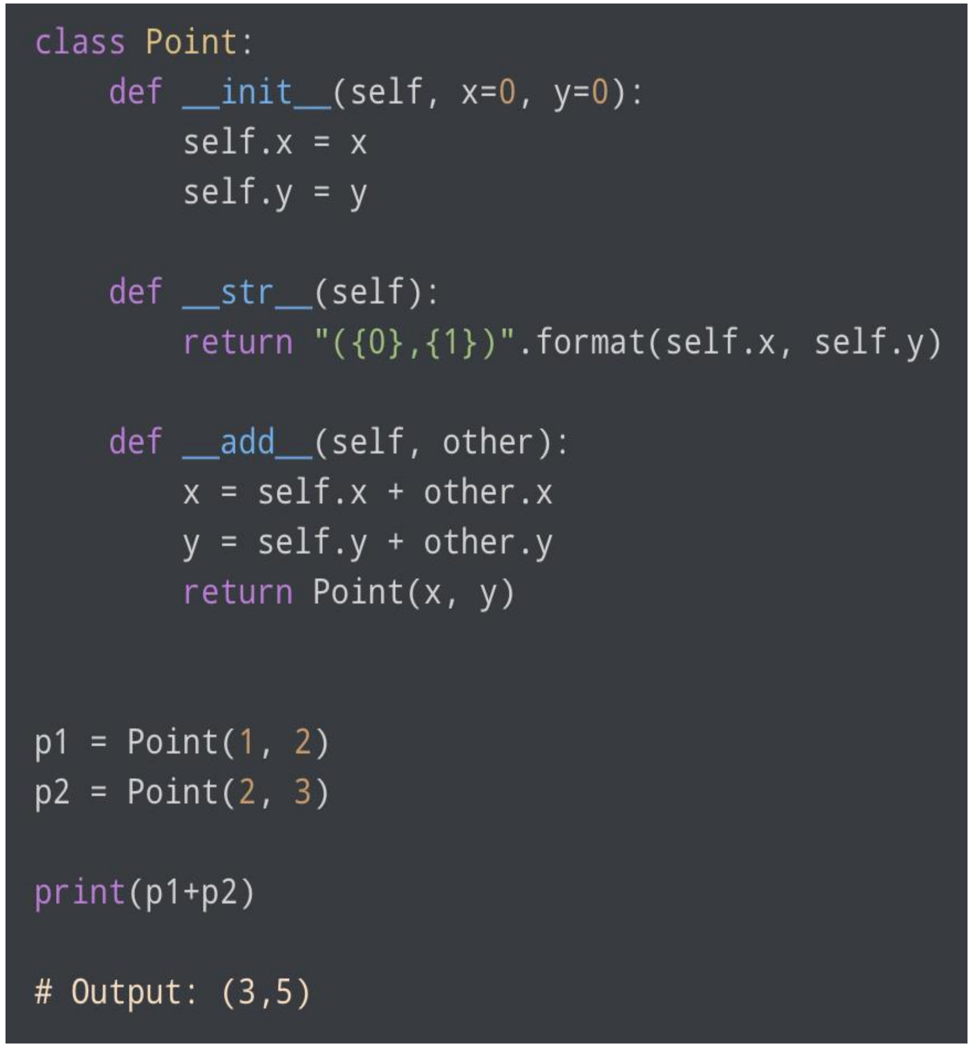
Function	Description
`__init__()`	initialize the attributes of the object
`__str__()`	returns a string representation of the object
`__len__()`	returns the length of the object
`__add__()`	adds two objects
`__call__()`	call objects of the class like a normal function

Example: + Operator Overloading in Python

To overload the + operator, we will need to implement __add__()

function in the class.

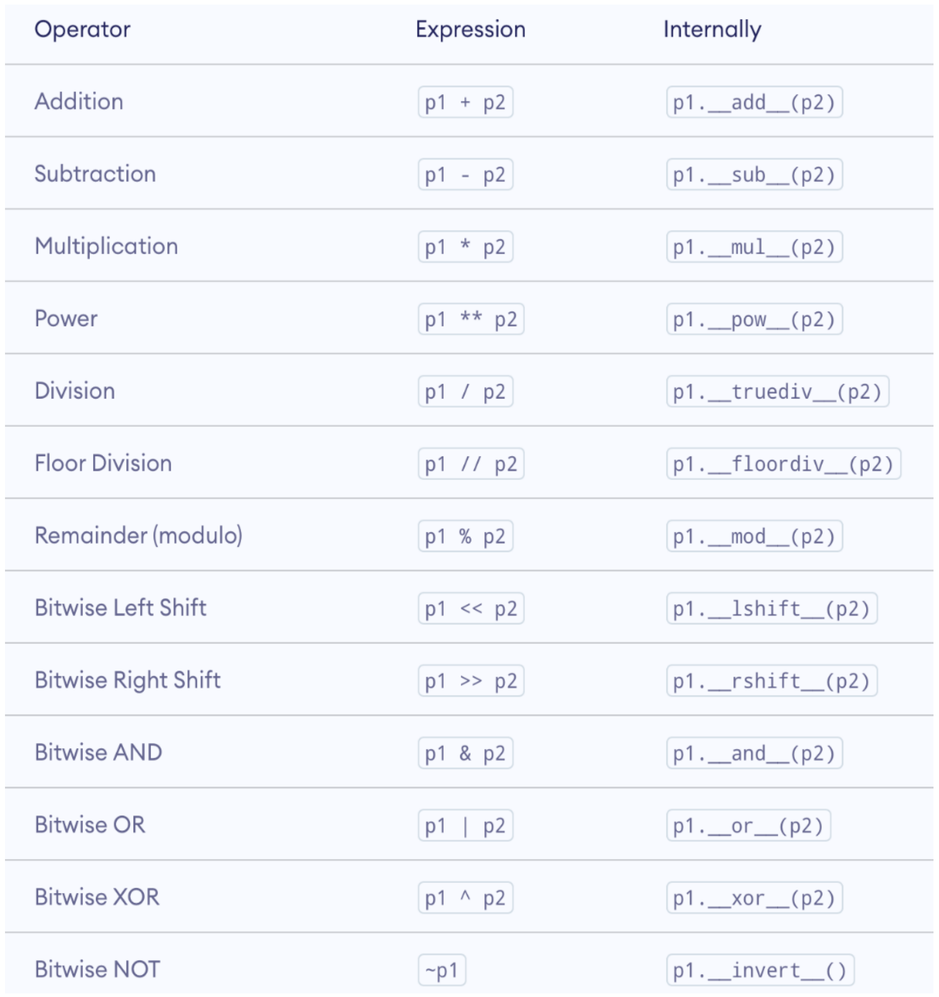
In the above example, what actually happens is that, when we use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified.



```python
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

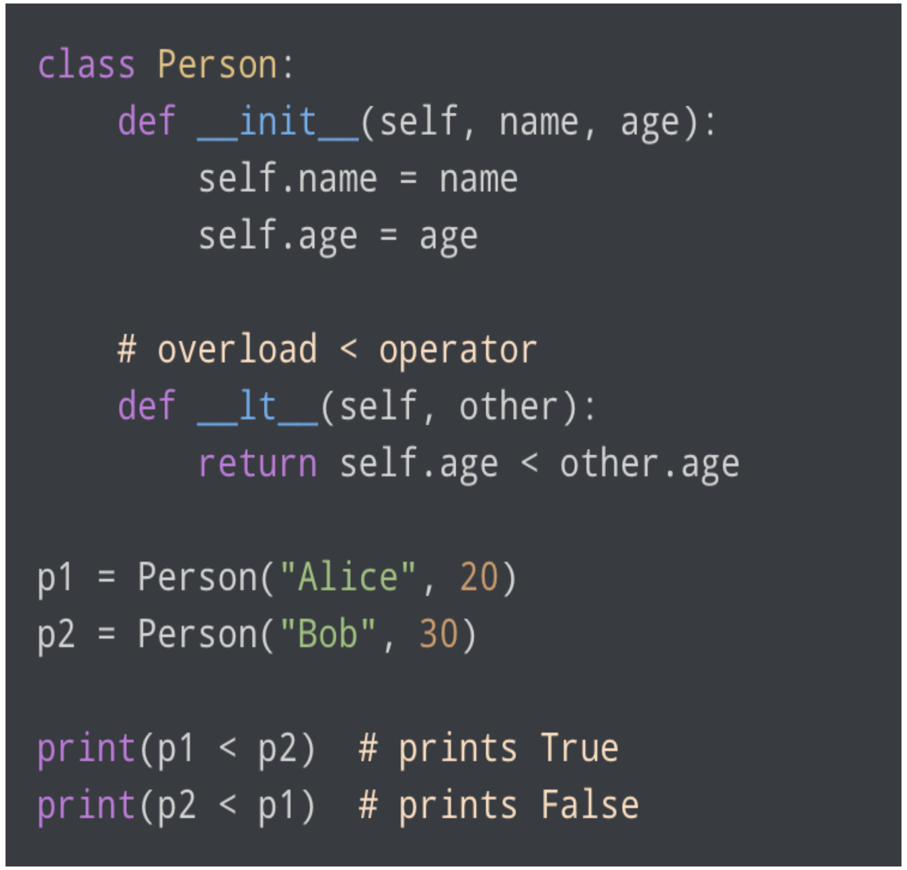
Operator Overloading

- Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.



Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Operator Overloading



```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # overload < operator
    def __lt__(self, other):
        return self.age < other.age

p1 = Person("Alice", 20)
p2 = Person("Bob", 30)

print(p1 < p2)  # prints True
print(p2 < p1)  # prints False
```

- Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.
- Here's an example of how we can overload the < operator to compare two objects the Person class based on their age:

Operator Overloading

Output:

```
True
False
```

- Here, __lt__() overloads the < operator to compare the age attribute of two objects.

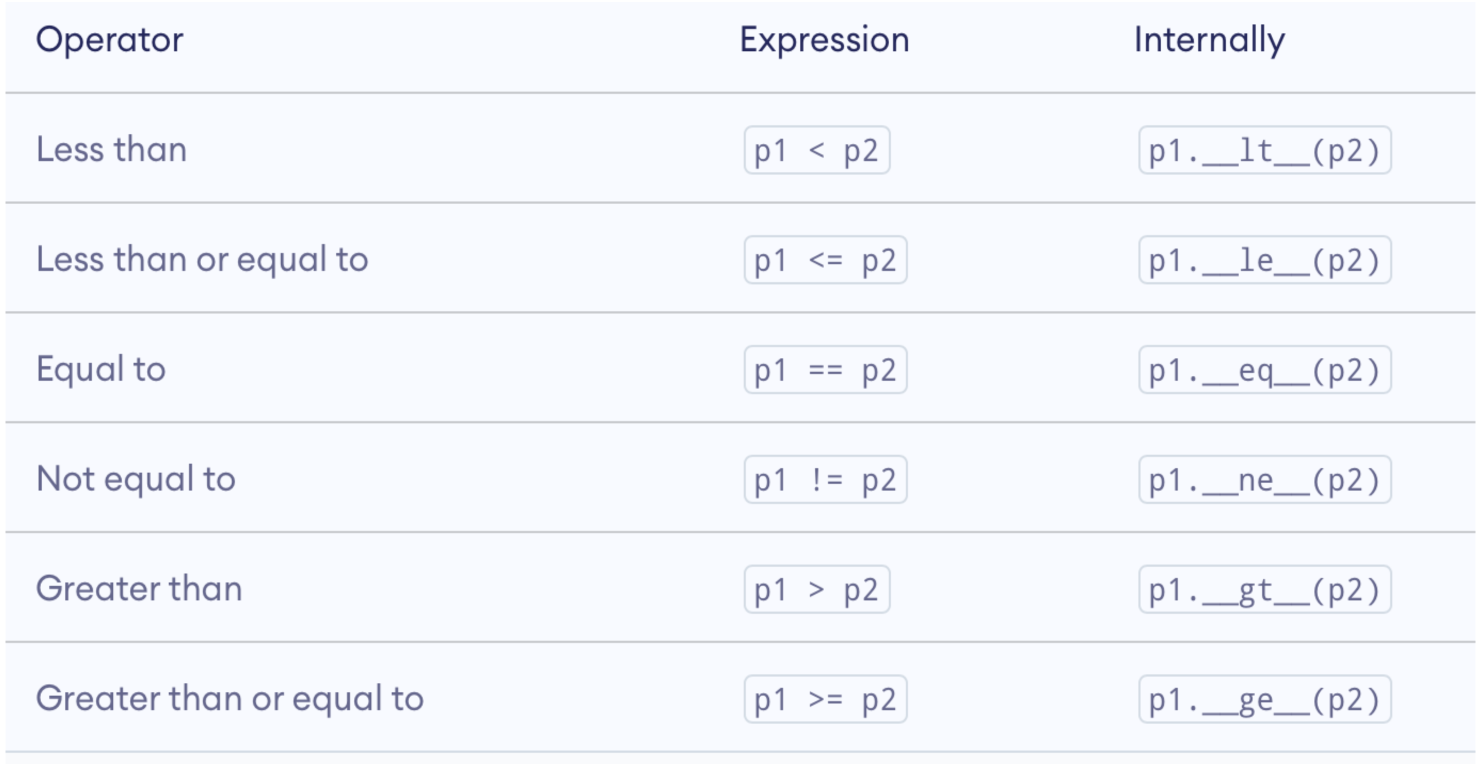
The __lt__() method returns:

- True - if the first object's age is less than the second object's age.

- False - if the first object's age is greater than the second object's age.

Operator Overloading

- Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below:



Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Activity - 11

Make a python module, which will ask the user to input the values in two different variables. On the basis of type of the user input values, overload the * operator in the following ways:

- **If the user input is int/float, then * operator will take the mean of the two numbers.**
- **If the user input is string, then * operator will merge the two stings.**
- **If the user enters one string value and another int/float value, thenn * operator will concatenate these two values.**

1. Control Flow Mechanisms

1. Control flow mechanisms

- Control flow mechanisms involve conditional statements

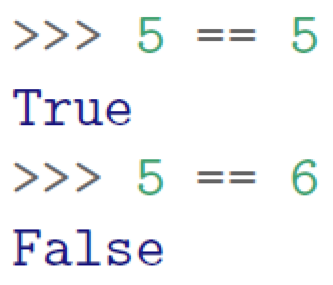
  1. (if-else),
  2. loops (for, while),
  3. and exception handling

  to control program execution.

Conditional execution : Boolean expressions

- A boolean expression is an expression that is either true or false

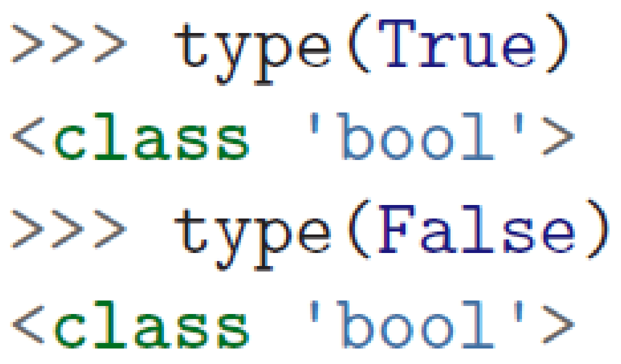
- The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:



```
>>> 5 == 5
True
>>> 5 == 6
False
```

Conditional execution : Boolean expressions

- True and False are special values that belong to the class bool; they are not strings:



```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Conditional execution : Boolean expressions

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

Logical operators

- There are three logical operators: and, or, and not

- The semantics (meaning) of these operators is similar to their meaning in English

- For example,

$$x > 0 \text{ and } x < 10$$

is true only if x is greater than 0 and less than 10

Logical operators

- n%2 == 0 or n%3 == 0 is true if *either of the conditions is true, that is, if the* number is divisible by 2 *or 3*

- Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y

Conditional execution

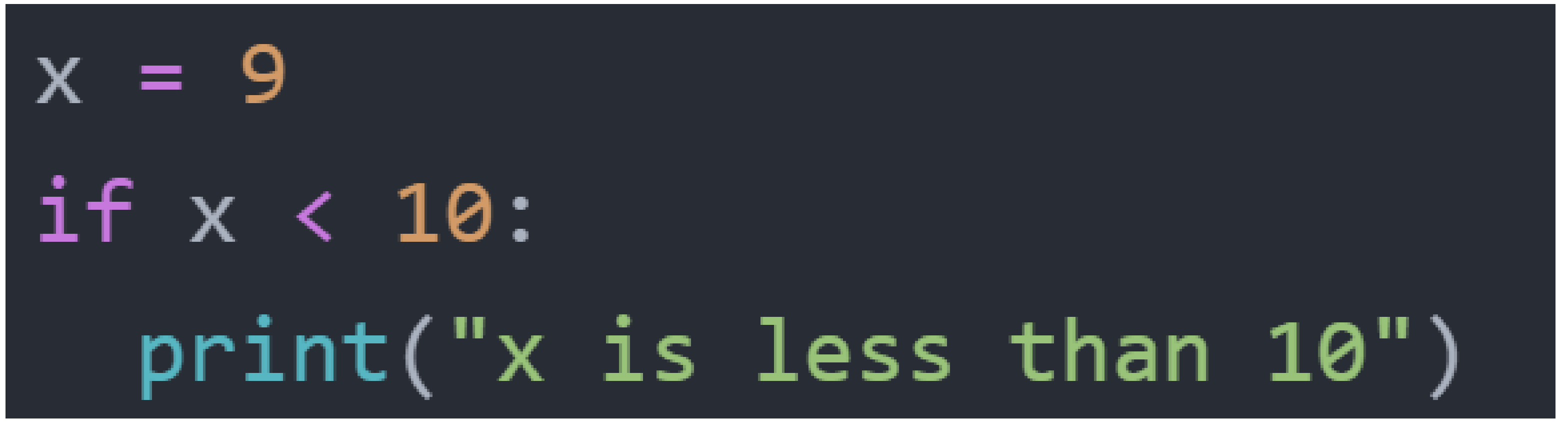
- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly

- Conditional statements give us this ability. The simplest form is the if statement

Conditional execution

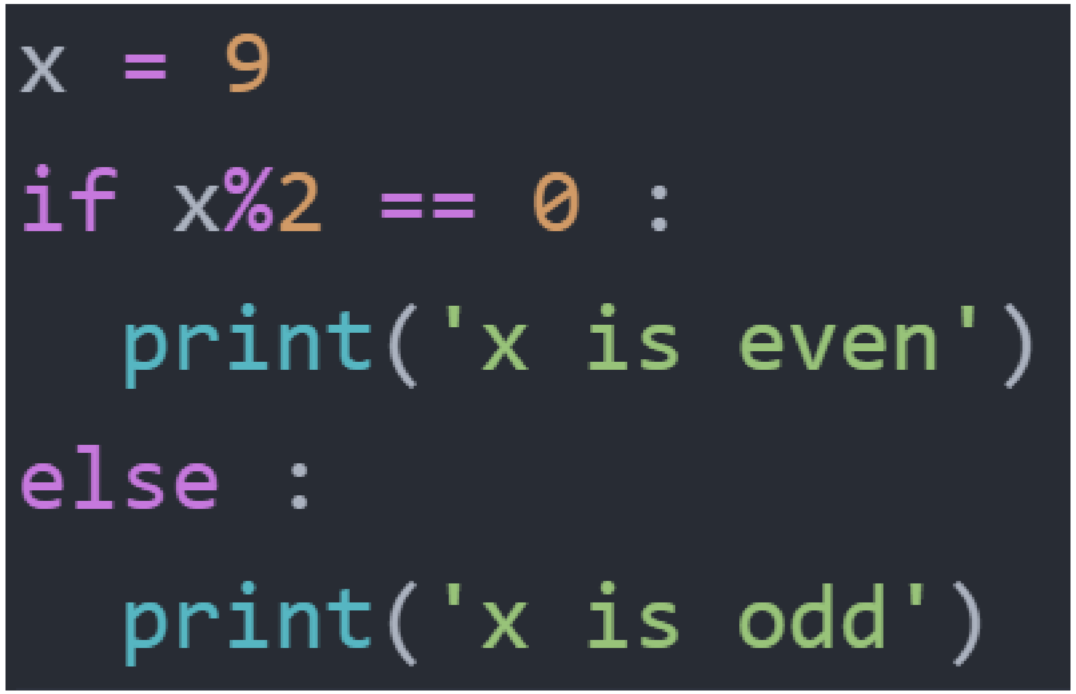
- if statements have the same structure as function definitions or for loops
- The statement consists of a header line that ends with the colon character (:) followed by an indented block
- Statements like this are called compound statements because they stretch across more than one line

Conditional execution



```
x = 9
if x < 10:
  print("x is less than 10")
```

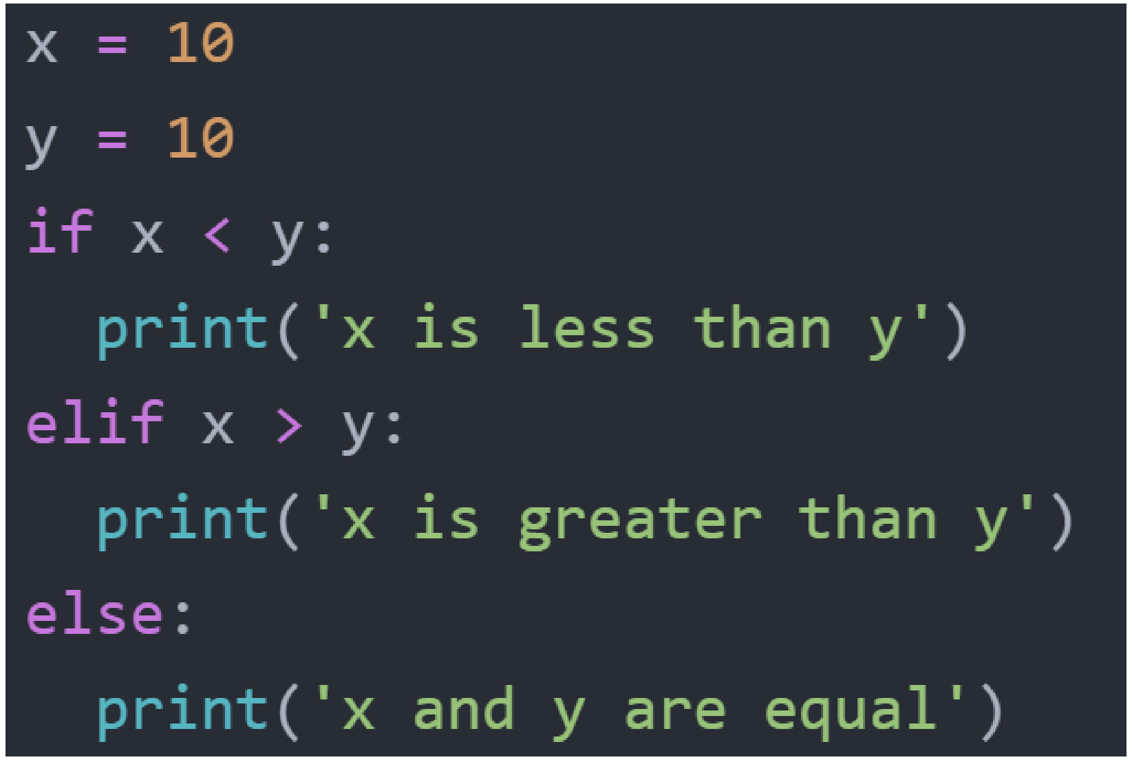
Alternative execution



```
x = 9
if x%2 == 0 :
  print('x is even')
else :
  print('x is odd')
```

Chained conditionals

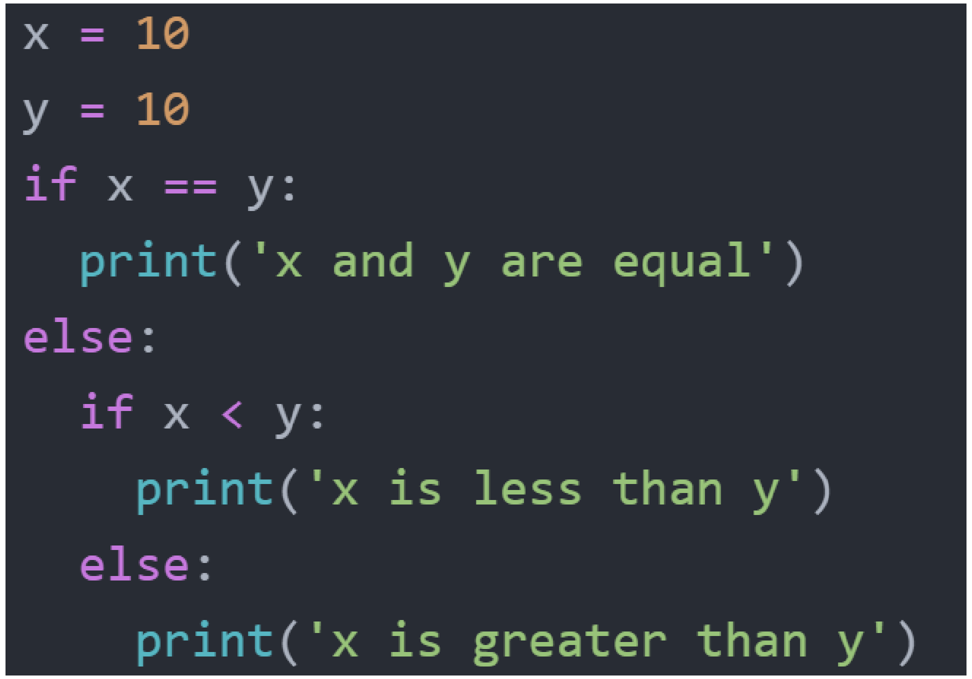
- Sometimes there are more than two possibilities and we need more than two branches



```python
x = 10
y = 10
if x < y:
  print('x is less than y')
elif x > y:
  print('x is greater than y')
else:
  print('x and y are equal')
```

Nested conditionals

- One conditional can also be nested within another. We could have written the three-branch example like this



```python
x = 10
y = 10
if x == y:
  print('x and y are equal')
else:
  if x < y:
    print('x is less than y')
  else:
    print('x is greater than y')
```

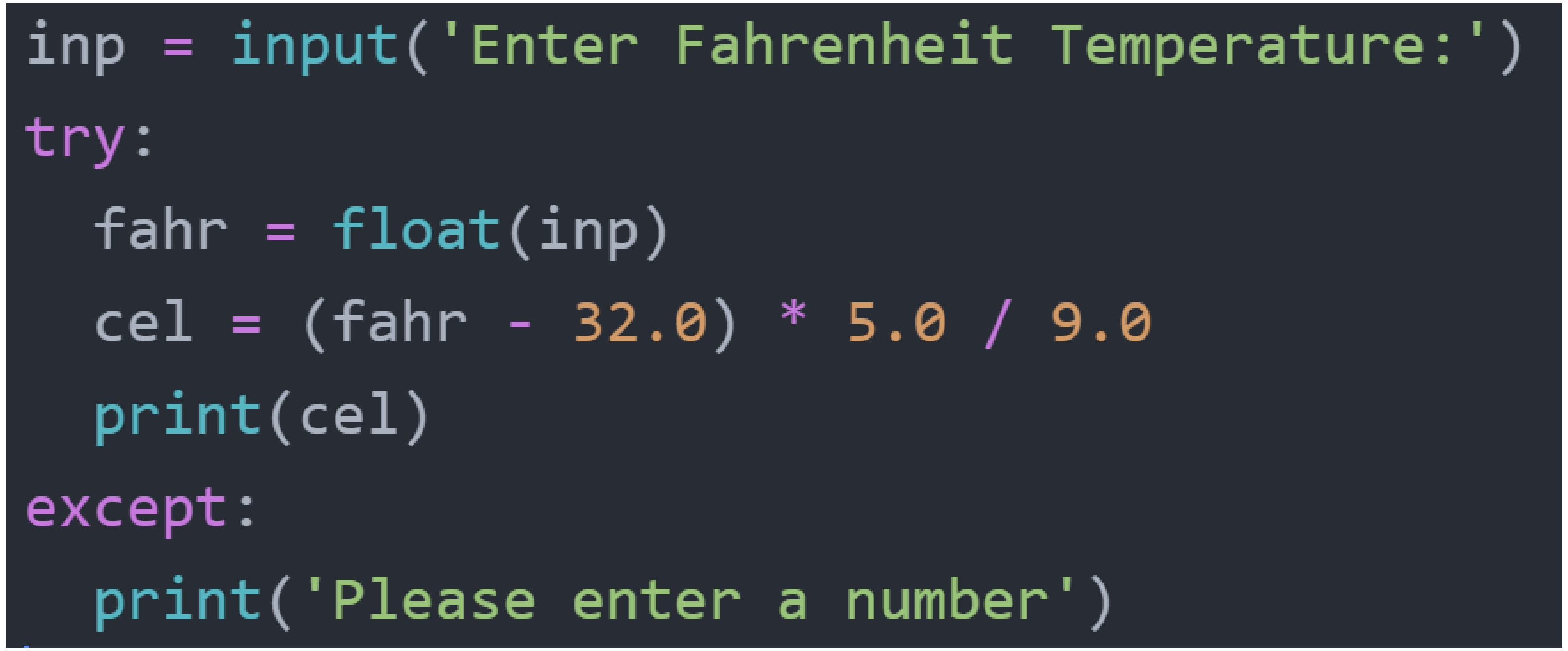
Catching exceptions using try and except

- There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”

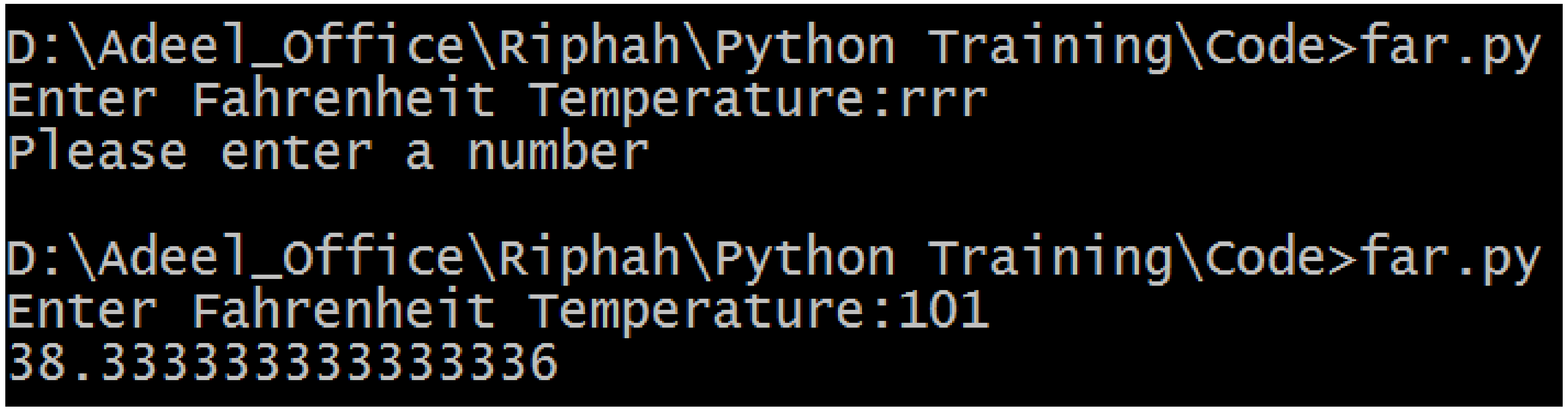
- The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs

- These extra statements (the except block) are ignored if there is no error

Catching exceptions using try and except



```python
inp = input('Enter Fahrenheit Temperature:')
try:
  fahr = float(inp)
  cel = (fahr - 32.0) * 5.0 / 9.0
  print(cel)
except:
  print('Please enter a number')
```



```
D:\Adeel_Office\Riphah\Python Training\Code>far.py
Enter Fahrenheit Temperature:rrr
Please enter a number

D:\Adeel_Office\Riphah\Python Training\Code>far.py
Enter Fahrenheit Temperature:101
38.333333333333336
```

Activity - 3

- Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

```
Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Activity - 4

- Write a program to prompt the user for hours and rate per hour using input to compute gross pay. Pay the hourly rate for the hours up to 40 and 1.5 times the hourly rate for all hours worked above 40 hours. Use 45 hours and a rate of 10.50 per hour to test the program (the pay should be 498.75). You should use **input** to read a string and **float()** to convert the string to a number. Do not worry about error checking the user input - assume the user types numbers properly

Activity - 5

- Rewrite your pay program using try and except so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows executions of the program
  - Enter Hours: forty
  - Error, please enter numeric input

Activity - 8

- Write a program to prompt the user for hours and rate per hour using input to compute gross pay. Pay should be the normal rate for hours up to 40 and time-and-a-half for the hourly rate for all hours worked above 40 hours

- Put the logic to do the computation of pay in a function called **computepay()** and use the function to do the computation

- The function should return a value. Use 45 hours and a rate of 10.50 per hour to test the program (the pay should be 498.75). You should use **input** to read a string and **float()** to convert the string to a number. Do not worry about error checking the user input unless you want to - you can assume the user types numbers properly. Do not name your variable sum or use the sum() function.

Iteration

Updating variables

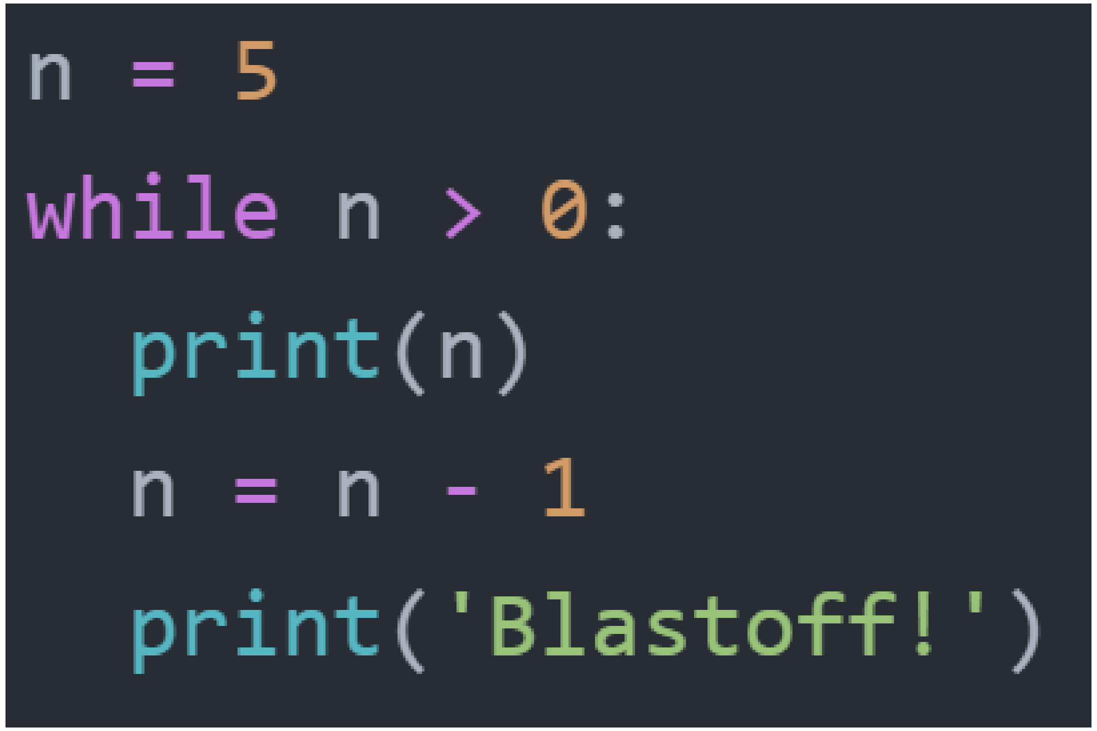
- A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old

```
x = x + 1
```

The while statement

- Computers are often used to automate repetitive tasks
- Repeating identical or similar tasks without making errors is something that computers do well and people do poorly
- One form of iteration in Python is the **while** statement

The while statement



```
n = 5
while n > 0:
  print(n)
  n = n - 1
  print('Blastoff!')
```

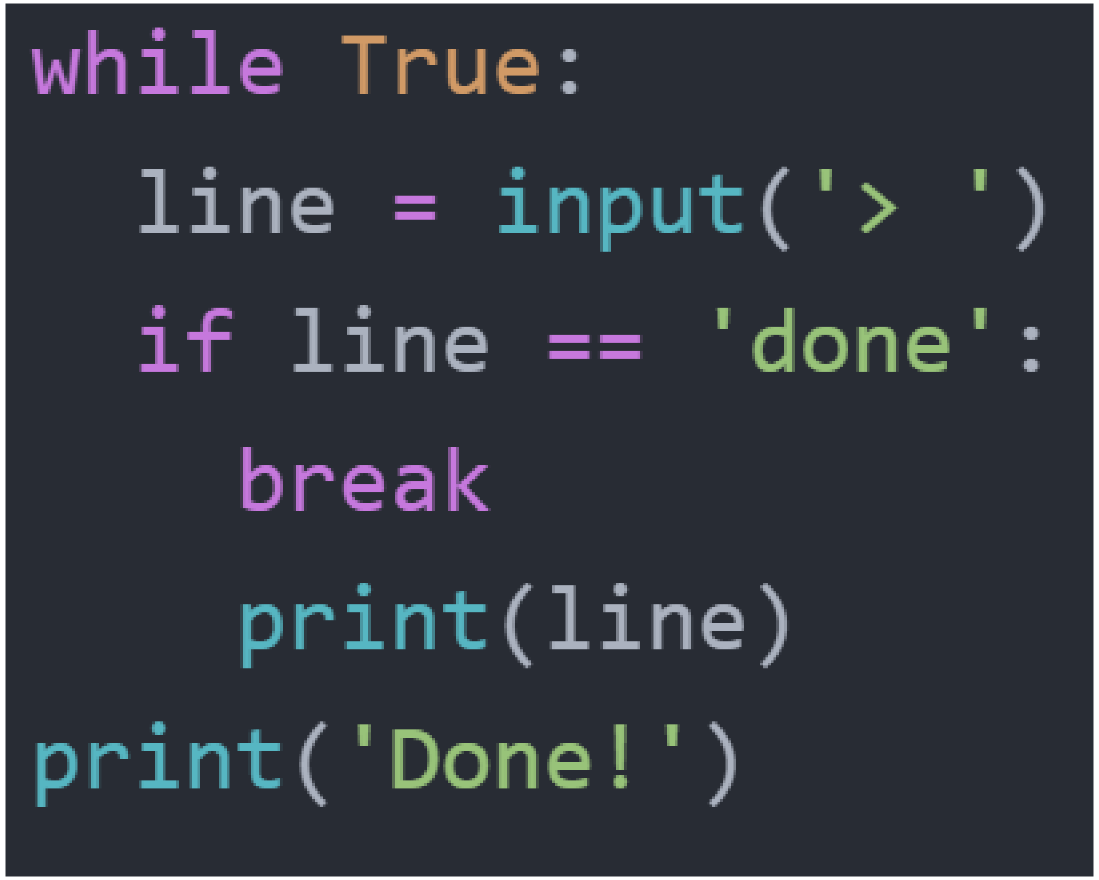
The while statement

- Evaluate the condition, yielding True or False
- If the condition is false, exit the while statement and continue execution at the next statement
- If the condition is true, execute the body and then go back to step 1

Infinite loops

- Sometimes you don’t know it’s time to end a loop until you get half way through the body
- In that case you can write an infinite loop on purpose and then use the **break** statement to jump out of the loop

Break Statement



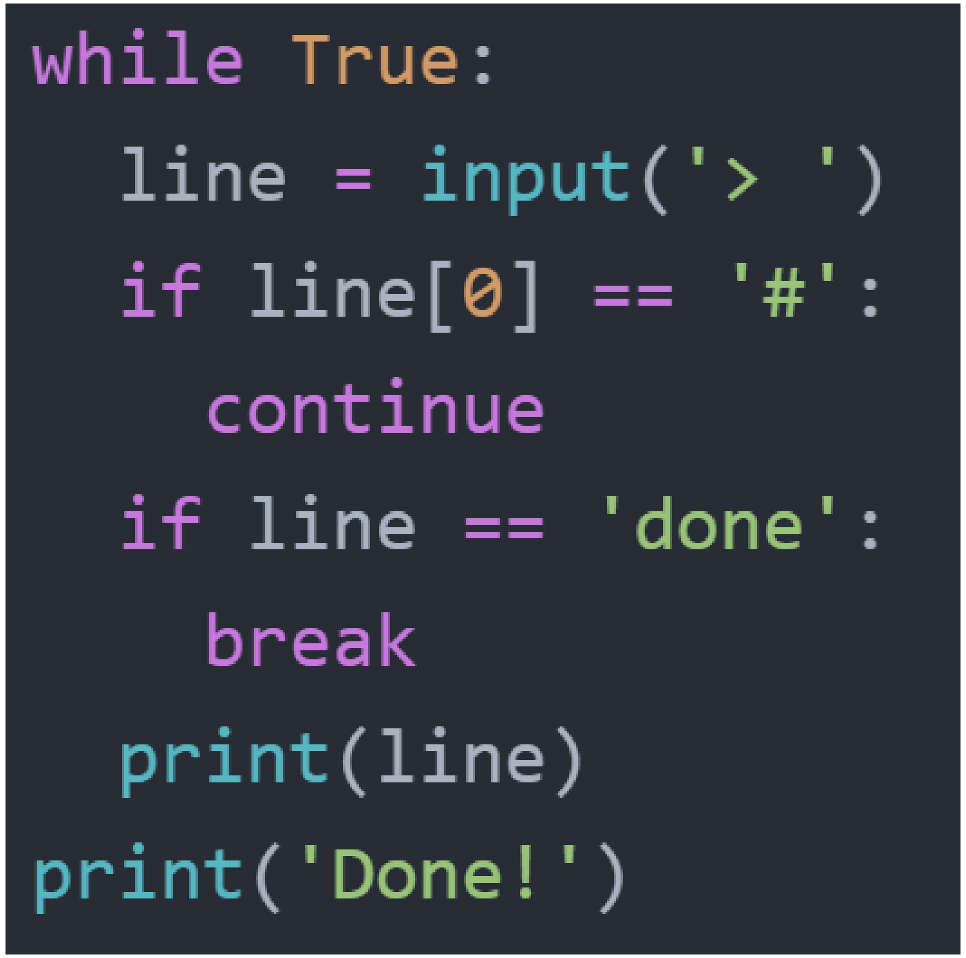
```
while True:
  line = input('> ')
  if line == 'done':
    break
    print(line)
print('Done!')
```

Finishing iterations with continue

- Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration

- In that case you can use the **continue** statement to skip to the next iteration without finishing the body of the loop for the current iteration

Continue Statement

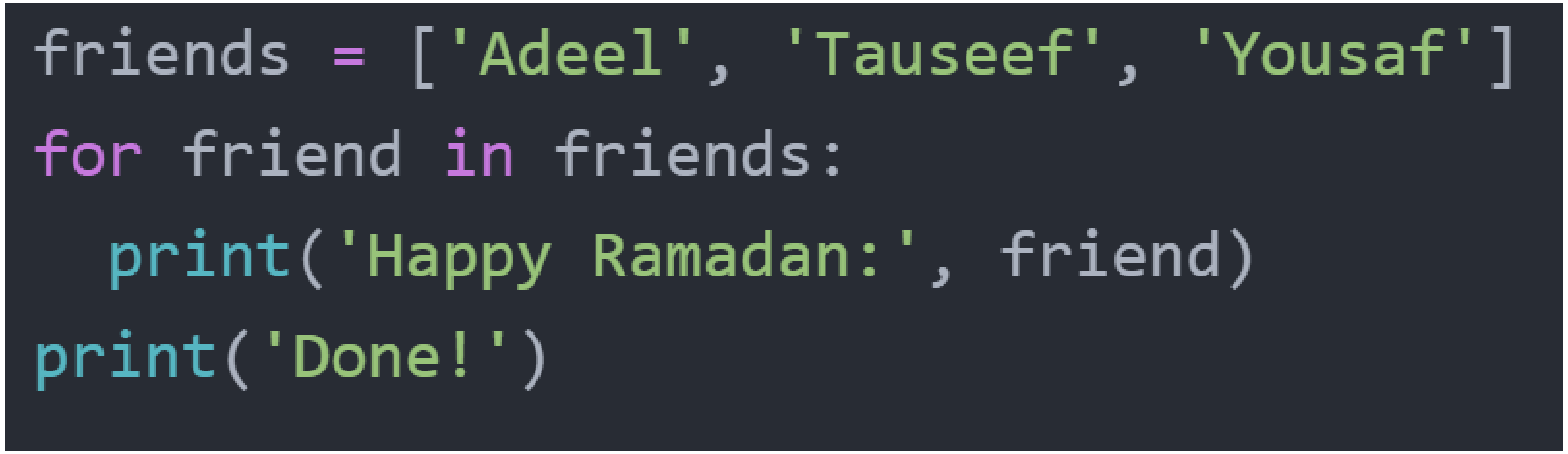


```
while True:
  line = input('> ')
  if line[0] == '#':
    continue
  if line == 'done':
    break
  print(line)
print('Done!')
```

Definite loops using for

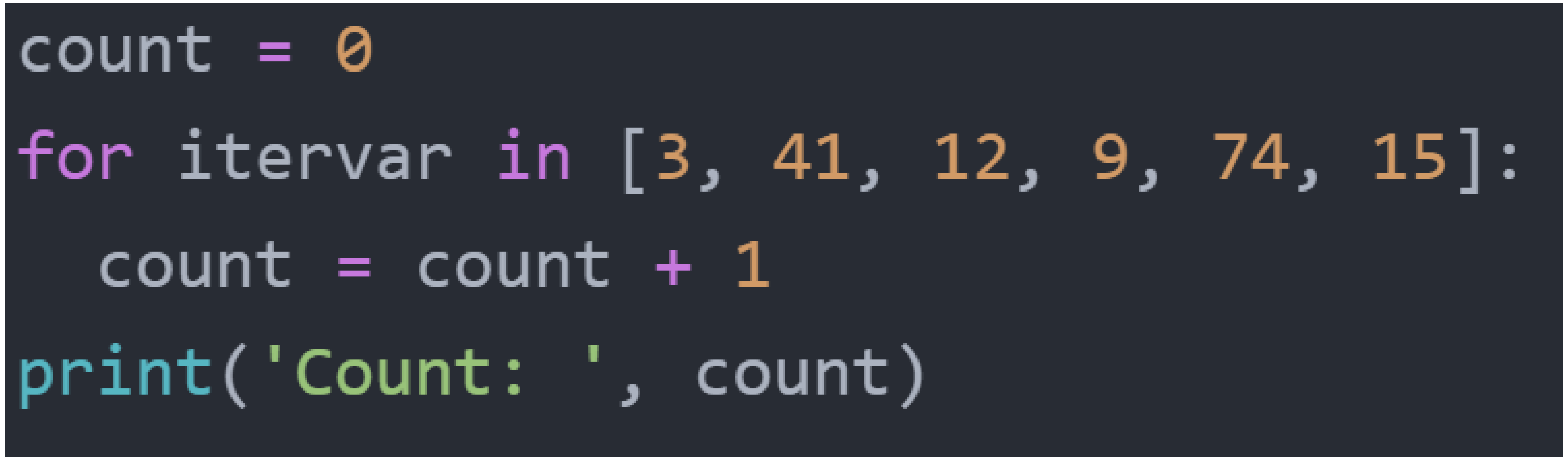
- Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers
- When we have a list of things to loop through, we can construct a definite loop using a for statement
- We call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set

for Loop



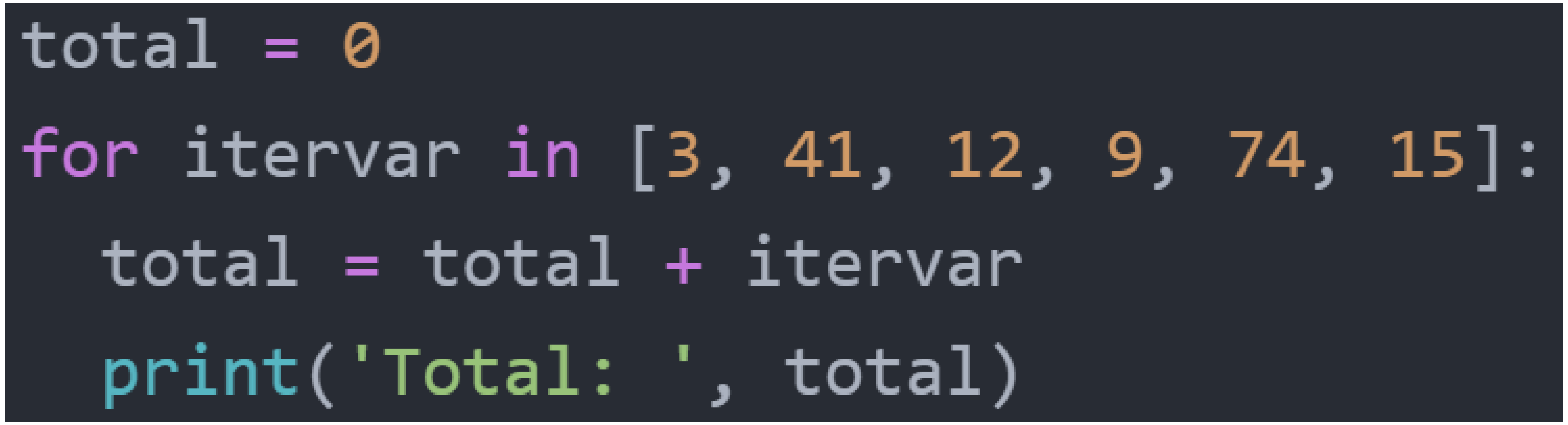
```python
friends = ['Adeel', 'Tauseef', 'Yousaf']
for friend in friends:
  print('Happy Ramadan:', friend)
print('Done!')
```

Counting and summing loops



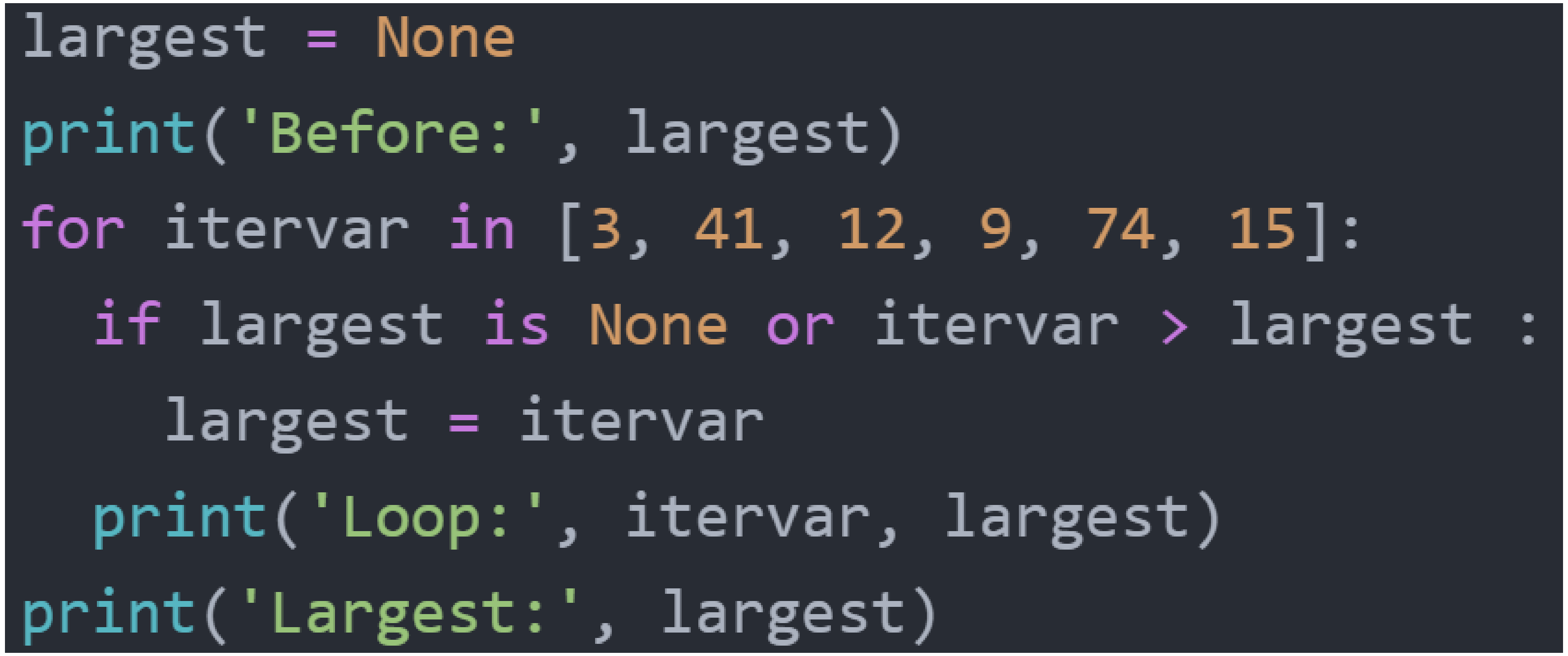
```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
  count = count + 1
print('Count: ', count)
```

Counting and summing loops



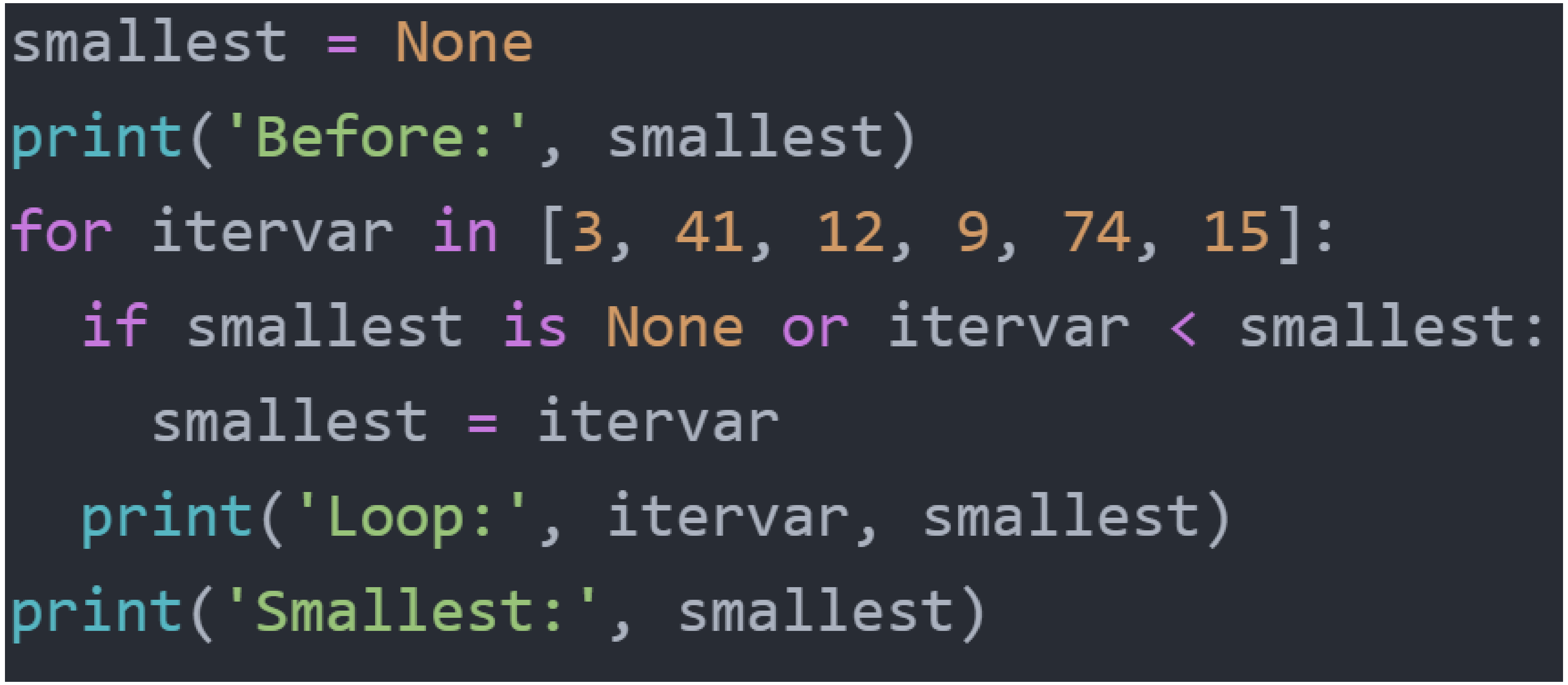
```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
  total = total + itervar
  print('Total: ', total)
```

Maximum and minimum loops



```python
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
  if largest is None or itervar > largest :
    largest = itervar
  print('Loop:', itervar, largest)
print('Largest:', largest)
```

Maximum and minimum loops



```python
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
  if smallest is None or itervar < smallest:
    smallest = itervar
  print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Maximum and minimum loops

- Again as in counting and summing, the built-in functions max() and min() make writing these exact loops unnecessary

- max([1,2,3,4])
- min([1,2,3,4])

Activity – 9.1

- **Write a program which repeatedly reads numbers until the user enters “done”. Once “done” is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using try and except and print an error message and skip to the next number.**

Python Iterables

- Anything that you can loop over in Python is called an iterable. For example, a list is an iterable. For an object to be considered an iterable, it must have the __iter()__ method.

```
numbers = [1, 4, 9]
print(dir(numbers))


'__iter__'
```

Python Iterables

- Iterator in Python is simply an object that can return data one at a time while iterating over it.
- For an object to be an iterator, it must implement two methods:

```
1. __iter__()
2. __next__()
```

Python Iterables

- We can see that there is a special __iter__() method among all these methods. Let's call this method.

```
numbers = [1, 4, 9]

value = numbers.__iter__()
print(value)


<list_iterator object at 0x7fa223878d00>
```

The next() method

- The __next__() method returns the next value in the iteration.

```
numbers = [1, 4, 9]
value = numbers.__iter__()

item1 = value.__next__()
print(item1)


1
```

- Here, we have already reached the end of our list. Now, let's see what happens if we further try to get the next value.

```
numbers = [1, 4, 9]
value = iter(numbers)

item1 = next(value)
print(item1)

item2 = next(value)
print(item2)

item3 = next(value)
print(item3)

item4 = next(value)
print(item4)
```

Output

Since our list had only 3 elements, the call to the fourth next() method raised the StopIteration exception.

```
1

4

9

Traceback (most recent call last):

  File "<string>", line 13, in <module>

StopIteration
```

Working of *for* loops

Did you know that for loops internally use the while loop to iterate through sequences?

```
num_list = [1, 4, 9]

iter_obj = iter(num_list)

while True:
    try:
        element = next(iter_obj)
        print(element)
    except StopIteration:
        break
```

Activity - 6

- Create Custom Iterators object to generate a sequence of even numbers such as 2, 4, 6, 8 and so on.

Solution

```
class Even:
    def __init__(self, max):
        self.n = 2
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= self.max:
            result = self.n
            self.n += 2
            return result
        else:
            raise StopIteration
```

```
numbers = Even(10)

print(next(numbers))
print(next(numbers))
print(next(numbers))
```

Python Generators

A generator is simply a function but with slight modification. In generator function, we use the yield keyword to get the next item of the iterator.

```python
def even_generator():
    n = 0
    n += 2
    yield n

    n += 2
    yield n

    n += 2
    yield n
```

```python
numbers = even_generator()

print(next(numbers))
print(next(numbers))
print(next(numbers))
```

Let's make this generator return even numbers till a certain max number.

Activity - 7

- Create Custom Generator which return even numbers till a certain max number

Solution

In this case, our generator could generate even numbers only till 4. So, using next() function for the third time raised a **StopIteration** exception.

```python
def even_generator(max):
    n = 2

    while n <= max:
        yield n
        n += 2
```

```python
numbers = even_generator(4)

print(next(numbers))
print(next(numbers))
print(next(numbers))
```

1. I/O operations / Files Handling and processing

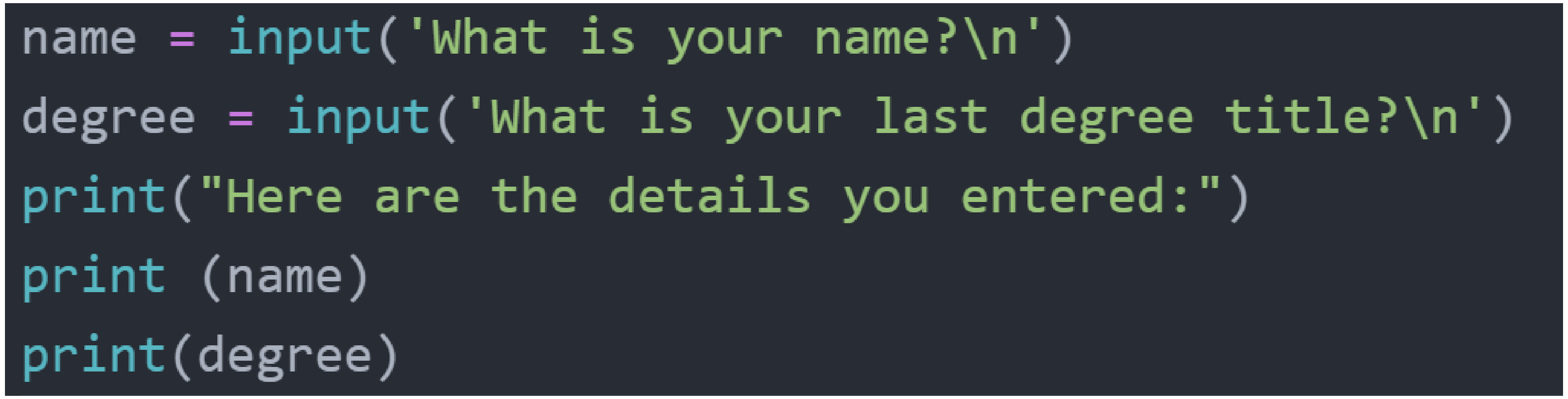
I/O Operations

- Refer to the processes of reading data from external sources (input) and writing data to external destinations (output). Python provides several built-in functions and libraries to perform I/O operations.
    - **Reading Input from the User**
    - **Reading from Files**
    - **Writing to Files**
    - **Reading Line by Line**
    - **Standard Streams**
    - **Redirecting Standard Streams**
    - **Reading and Writing Binary Data**

Asking the user for input

- Sometimes we would like to take the value for a variable from the user via their keyboard
- Python provides a built-in function called **input** that gets input from the keyboard1
- When this function is called, the program stops and waits for the user to type something
- When the user presses Return or Enter, the program resumes and input returns what the user typed as a string

Asking the user for input



```python
name = input('What is your name?\n')
degree = input('What is your last degree title?\n')
print("Here are the details you entered:")
print (name)
print(degree)
```

Escape Sequences

- The sequence \n at the end of the prompt represents a *newline, which is a special* character that causes a line break

- That’s why the user’s input appears below the prompt

Conversion

- If you expect the user to type an integer, you can try to convert the return value to int using the int() function

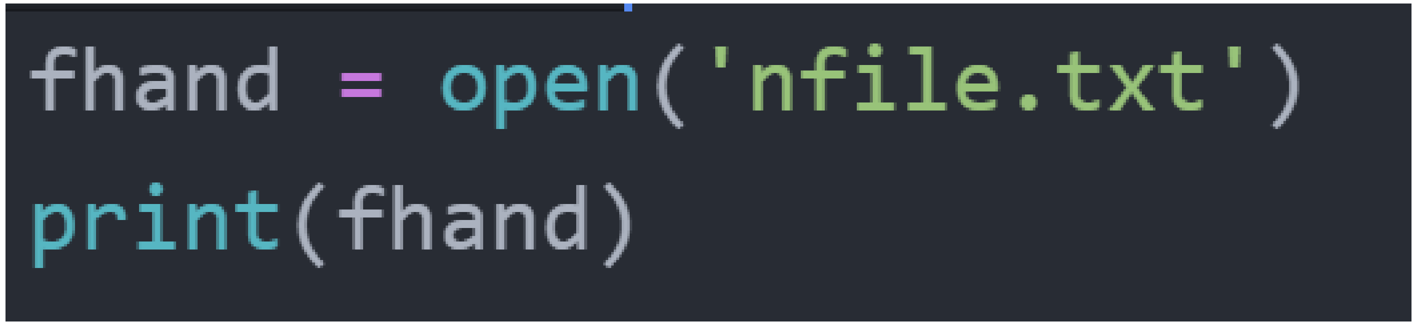
- For Example

  - int(input())
  - float(input())

Opening files

- When we want to read or write a file
- Opening the file communicates with your operating system, which knows where the data for each file is stored
- When you open a file, you are asking the operating system to find the file by name and make sure the file exists

Opening files

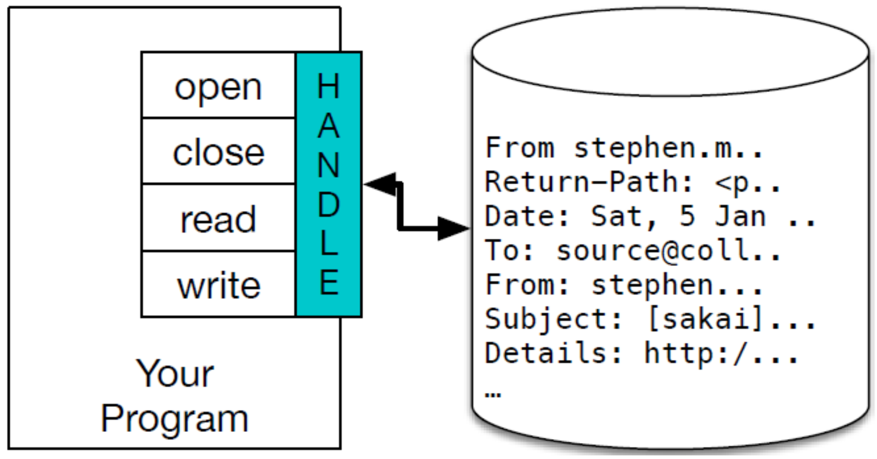


```
fhand = open('nfile.txt')
print(fhand)
```

```
<_io.TextIOWrapper name='nfile.txt'
mode='r' encoding='cp1252'>
```

File handle

- If the open is successful, the operating system returns us a file handle
- The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data



Reading from Files

- We can open and read files using the **open()** function and various file modes ('r' for reading, 'w' for writing, 'a' for appending, etc.).

```
# Reading from a file
with open(“nfile.txt", "r") as file:
    content = file.read()
    print(content)
```

Writing to Files

- To write data to a file, use the 'w' mode with the **open()** function.

```
# Writing to a file
with open("output.txt", "w") as file:
        file.write("This is some data to write to the file.")
```

Reading Line by Line

- We can also read a file line by line using a **for** loop.

```
# Reading a file line by line
with open("example.txt", "r") as file:
        for line in file:
                print(line)
```

Standard Streams

- Python has three standard streams:

  - sys.stdin: Standard input stream (keyboard).
  - sys.stdout: Standard output stream (console).
  - sys.stderr: Standard error stream (console).

You can use these streams for input and output operations.

sys.stdin

- sys.stdin stream is used for standard input, typically reading input from the keyboard.

```
import sys
print("Enter your name:")
user_input = sys.stdin.readline()
user_input = user_input.strip()
print("Hello, " + user_input + "! You entered your name.")
```

sys.stdout

- sys.stdout stream is used for standard output, typically displaying output to the console.

```
import sys
with open("output.txt", "w") as file:
	sys.stdout = file
	print("This will be written to output.txt")
```

sys.stderr

- sys.stderr stream is used for standard error output, typically used for displaying error messages or exceptions.

```
import sys
try:
    result = 10 / 0
except ZeroDivisionError as e:
    sys.stderr.write("Error: Division by zero!\n")
```

Reading and Writing Binary Data

- You can read and write binary data using the 'rb' and 'wb' modes when working with files.

```
# Reading binary data
with open("binary_file.bin", "rb") as file:
        binary_data = file.read()
# Writing binary data
with open("output.bin", "wb") as file:
        file.write(binary_data)
```

Activity - 7

- **Write a program to read through a file and print the contents of the file (line by line) all in upper case. Executing the program will look as follows:**

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
SAT, 05 JAN 2008 09:14:16 -0500
```

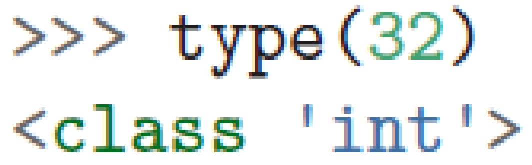
Note: You need to first create this file in the same directory

2. Functions

- a. built-in
- b. user-defined functions
- c. interaction between functions
- d. generators
- e. recursion

Functions : Function calls

- In the context of programming, a function is a named sequence of statements that performs a computation
- When you define a function, you specify the name and the sequence of statements
- Later, you can "call" the function by name



```
>>> type(32)
<class 'int'>
```

Functions

- The name of the function is type
- The expression in parentheses is called the argument of the function
- The argument is a value or variable that we are passing into the function as input to the function

Functions

- The result, for the type function, is the type of the argument
- It is common to say that a function “takes” an argument and “returns” a result
- The result is called the return value

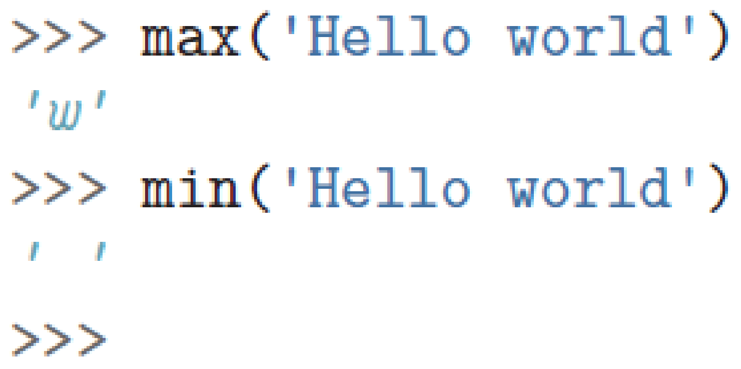
Built-in functions

- Python provides a number of important built-in functions that we can use without needing to provide the function definition

- The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use

Built-in functions : Max () – Min ()

- The max and min functions give us the largest and smallest values in a list, respectively:

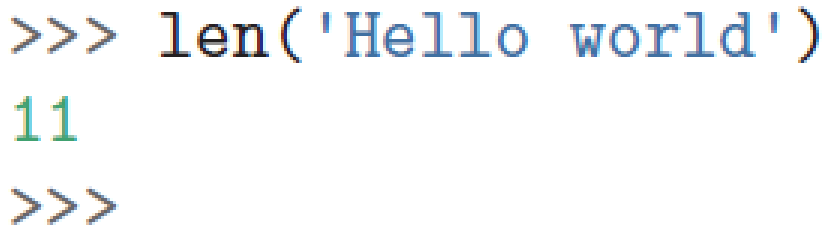


```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Built-in functions : len()

- Another very common built-in function is the len function which tells us how many items are in its argument

- If the argument to len is a string, it returns the number of characters in the string

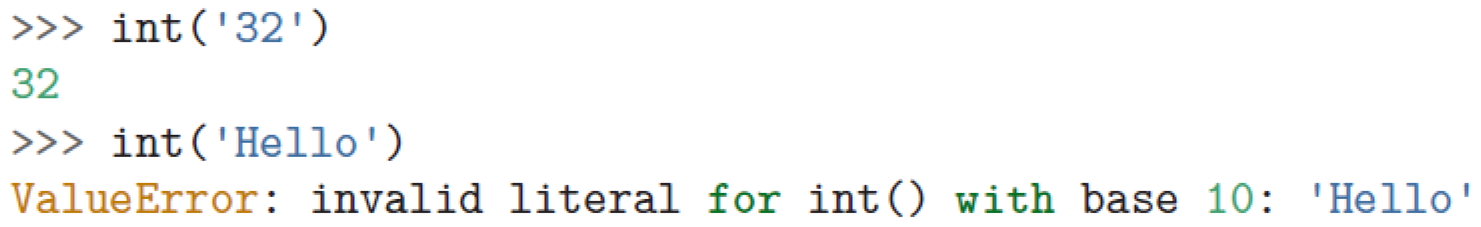


```
>>> len('Hello world')
11
>>>
```

Type conversion functions

- Python also provides built-in functions that convert values from one type to another

- The int function takes any value and converts it to an integer, if it can, or complains otherwise:



```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

Type conversion functions

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Math functions

- Python has a math module that provides most of the familiar mathematical functions

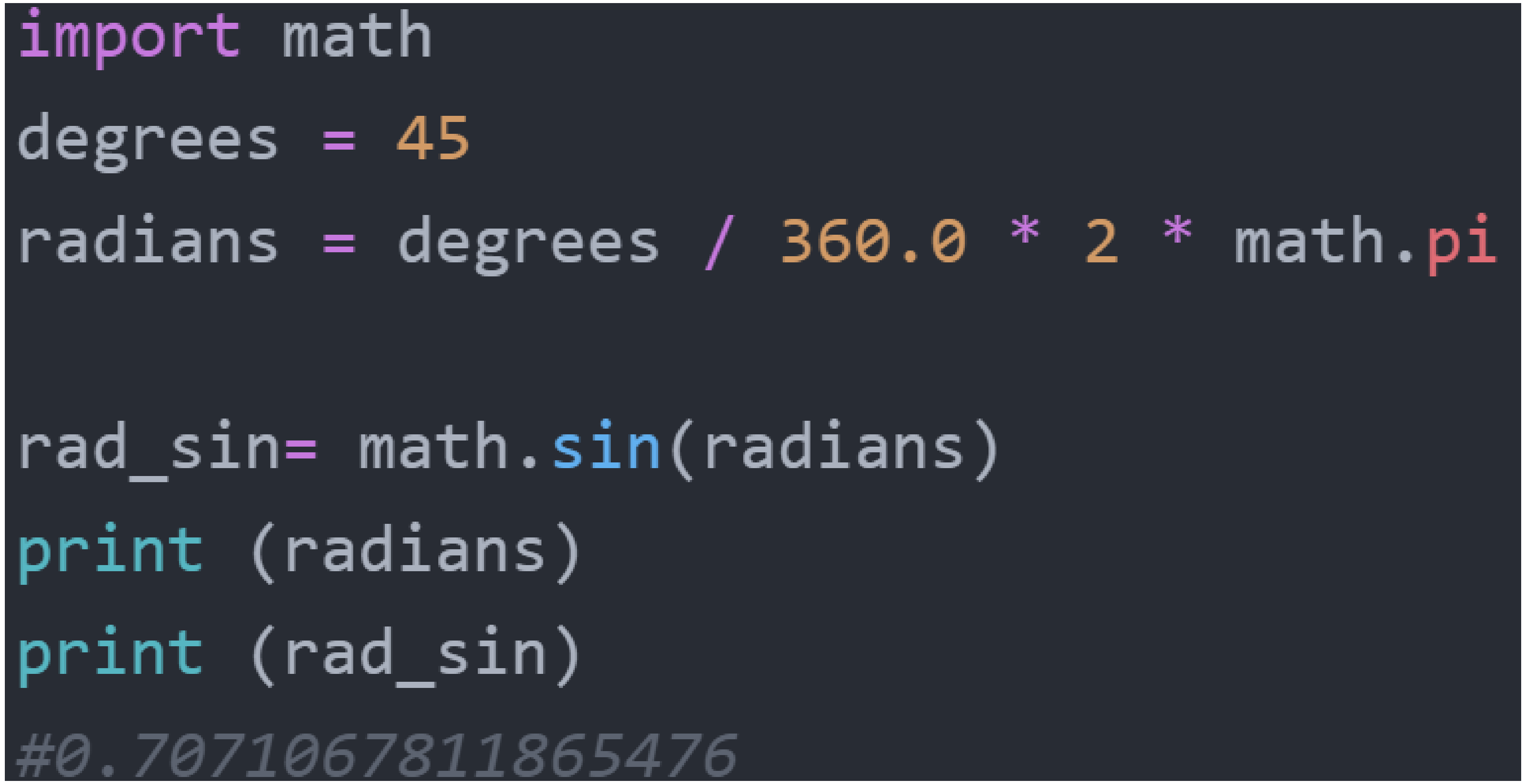
- Before we can use the module, we have to import it:

```
>>> import math
```

Dot Notation

- The module object contains the functions and variables defined in the module
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period)
- This format is called dot notation

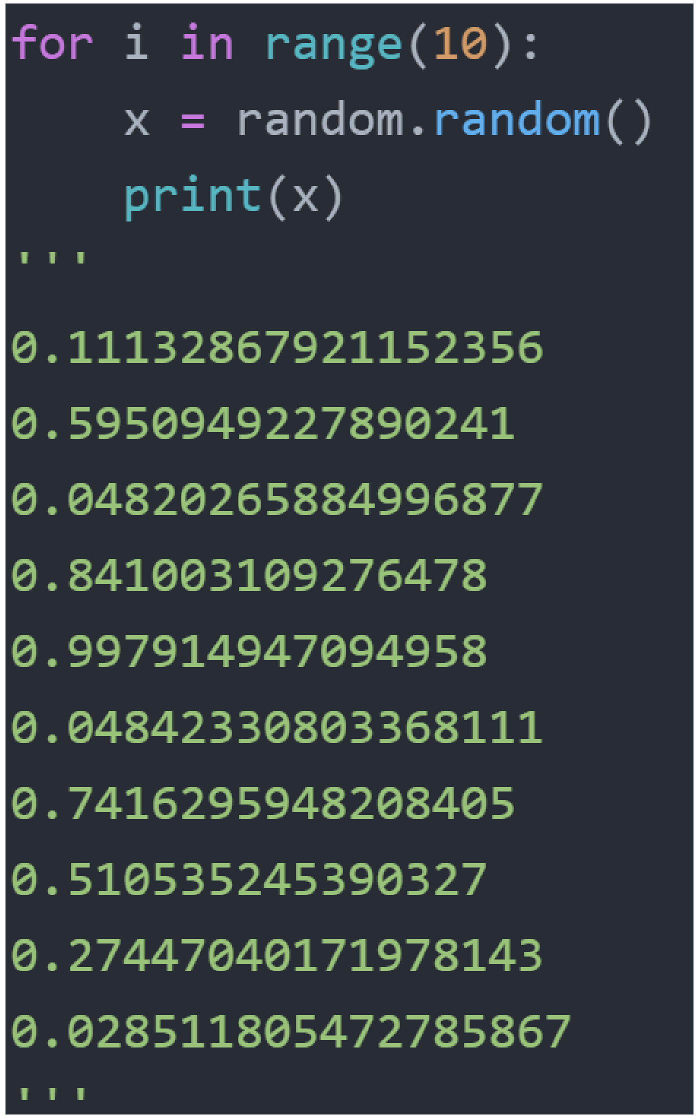
Dot Notation



```python
import math
degrees = 45
radians = degrees / 360.0 * 2 * math.pi

rad_sin= math.sin(radians)
print (radians)
print (rad_sin)
#0.7071067811865476
```

Random numbers

- The random module provides functions that generate pseudorandom numbers



```python
for i in range(10):
    x = random.random()
    print(x)
'''
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
'''
```

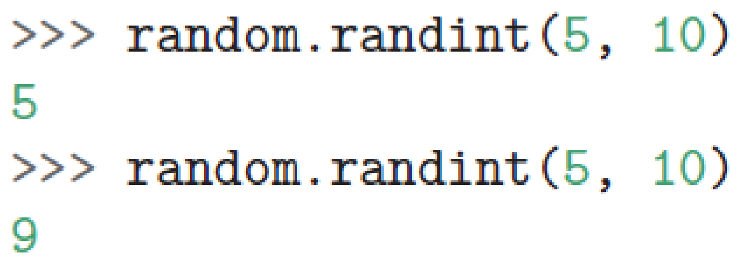
Activity- 8

- **Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.**

randint()

- The random function is only one of many functions that handle random numbers

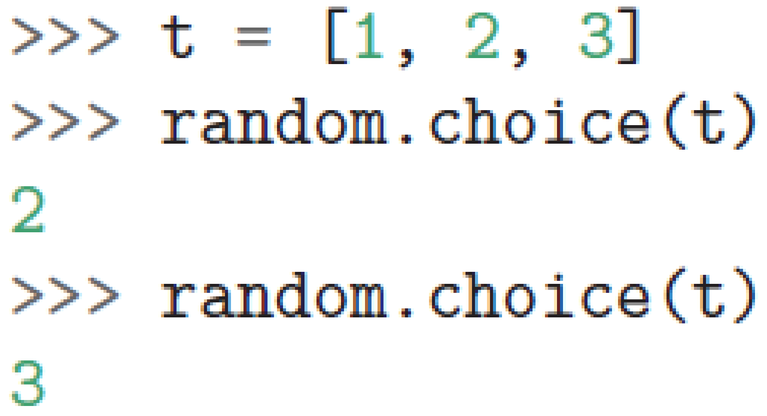
- The function randint takes the parameters low and high, and returns an integer between low and high (including both)



```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

random.choice()

- To choose an element from a sequence at random, you can use choice:



```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Rules to create functions

- def is a keyword that indicates that this is a function definition
- The name of the function is print_lyrics
- The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but thefirst character can’t be a number
- You can’t use a keyword as the name of a function, and you should avoid having a variable and a function with the same name

Adding new functions

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions

- A function definition specifies the name of a new function and the sequence of statements that execute when the function is called

- Once we define a function, we can reuse the function over and over throughout our program

Adding new functions

We can create our own functions in Python using the def keyword.

```
def function_name(parameters):
        # Function body
        # ...
        return result

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

Interaction Between Functions

Functions can interact with each other by calling one function from another. This allows us to break down complex tasks into smaller, more manageable functions.

```
def square(x):
        return x * x
def cube(x):
        return x * x * x
def square_and_cube(x):
        squared = square(x)
        cubed = cube(x)
        return squared, cubed
```

Generators

Generators are a way to create iterators in Python. They allow you to iterate over a potentially large sequence of data without storing it all in memory at once. You can define a generator using a function with the yield keyword.

```
def countdown(n):
	while n > 0:
		yield n
		n -= 1
for value in countdown(5):
	print(value)
```

Recursion

Recursion is a technique in which a function calls itself to solve a problem. It's useful for solving problems that can be broken down into smaller, similar subproblems.

```
def factorial(n):
        if n == 0:
                return 1
        else:
                return n * factorial(n - 1)
print(factorial(3))
```