MODULE II - Modularization and OOPs Concept

Functions: With and without argument, with and without return, recursive function, Date function, Math function, Lambda – Error handling – Classes and Objects – Inheritance – Polymorphism – Exception Handling

Functions: With and without argument, with and without return,

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Basic Concept: Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions*.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

Live Demo

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result –

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

Live Demo

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Recursion Function

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Recursion

What is recursion?

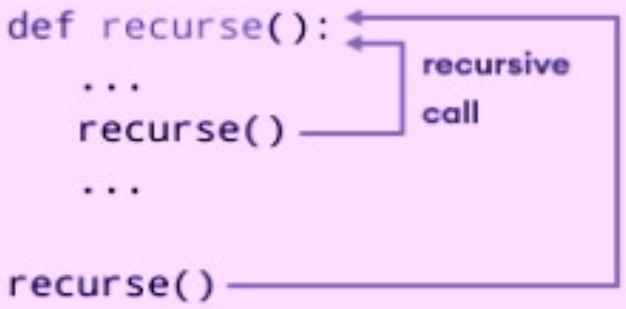
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called `recurse`.



Recursive Function in Python

Recursion

Example of a recursive function

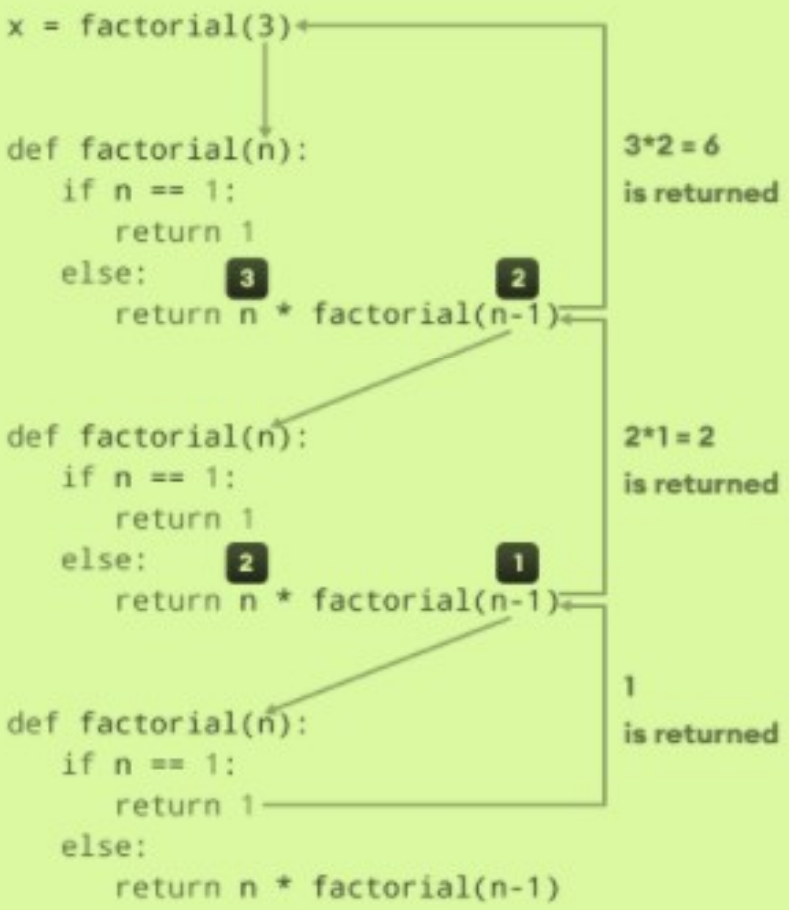
```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```

Recursion

Let's look at an image that shows a step-by-step process of what is going on:



Working of a recursive factorial function

Recursion

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is `1000`. If the limit is crossed, it results in `RecursionError`. Let's look at one such condition.

```
def recursor():
    recursor()
recursor()
```

Output

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

Date function, Math function, Lambda

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Date function

Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

Example

Import the datetime module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

Date function

Date Output

When we execute the code from the example above the result will be:

```
2021-06-10 21:22:16.019653
```

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

Date function

Creating Date Objects

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

Date function

The strftime() Method

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

Standard mathematical functions

The standard `math` module provides much of the functionality of a scientific calculator. Table 6.1 lists only a few of the available functions.

`mathfunctions` **Module**	
`sqrt`	Computes the square root of a number: `sqrt(x)` $= \sqrt{x}$
`exp`	Computes e raised a power: `exp(x)` $= e^x$
`log`	Computes the natural logarithm of a number: `log(x)` $= \log_e x = \ln x$
`log10`	Computes the common logarithm of a number: `log(x)` $= \log_{10} x$
`cos`	Computes the cosine of a value specified in radians: `cos(x)` $= \cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`pow`	Raises one number to a power of another: `pow(x,y)` $= x^y$
`degrees`	Converts a value in radians to degrees: `degrees(x)` $= \frac{\pi}{180} x$
`radians`	Converts a value in degrees to radians: `radians(x)` $= \frac{180}{\pi} x$
`fabs`	Computes the absolute value of a number: `fabs(x)` $= \lvert x \rvert$

prime number program using Maths functions

Listing 6.4: moreefficientprimes.py

```
1  from math import sqrt
2
3  max_value = eval(input('Display primes up to what value? '))
4  value = 2  #  Smallest prime number
5
6  while value <= max_value:
7      #  See if value is prime
8      is_prime = True  # Provisionally, value is prime
9      #  Try all possible factors from 2 to value - 1
10     trial_factor = 2
11     root = sqrt(value)
12     while trial_factor <= root:
13         if value % trial_factor == 0:
14             is_prime = False;    #  Found a factor
15             break                # No need to continue; it is NOT prime
16         trial_factor += 1        # Try the next potential factor
17     if is_prime:
18         print(value, end= ' ')   #  Display the prime number
19     value += 1                   # Try the next potential prime number
20
21 print()  #  Move cursor down to next line
```

Time function

- The time package contains a number of functions that relate to time. We will consider two: clock and sleep.
- The clock function allows us measure the time of parts of a program's execution. The clock returns a floating-point value representing elapsed time in seconds.
- On Unix-like systems (Linux and Mac OS X), clock returns the numbers of seconds elapsed since the program began executing.
- Under Microsoft Windows, clock returns the number of seconds since the first call to clock.

Time function : clock example 1

- timeit.py measures how long it takes a user to enter a character from the keyboard.

Listing 6.5: timeit.py

```
1 from time import clock
2
3 print("Enter your name: ", end="")
4 start_time = clock()
5 name = input()
6 elapsed = clock() - start_time
7 print(name, "it took you", elapsed, "seconds to respond")
```

The following represents the program's interaction with a particularly slow typist:

```
Enter your name: Rick
Rick it took you 7.246477029927183 seconds to respond
```

Time function : clock example 2

- measureprimespeed.py measures how long it takes a program to count all the prime numbers up to 10,000
- On one system, the program took about 1.25 seconds, on average, to count all the prime numbers up to 10,000.

Listing 6.7: measureprimespeed.py

```
1  from time import clock
2
3  max_value = 10000
4  count = 0
5  start_time = clock()    #  Start timer
6  #  Try values from 2 (smallest prime number) to max_value
7  for value in range(2, max_value + 1):
8      #  See if value is prime
9      is_prime = True  # Provisionally, value is prime
10     #  Try all possible factors from 2 to value - 1
11     for trial_factor in range(2, value):
12         if value % trial_factor == 0:
13             is_prime = False     #  Found a factor
14             break                # No need to continue; it is NOT prime
15     if is_prime:
16         count += 1               # Count the prime number
17 print()  #  Move cursor down to next line
18 elapsed = clock() - start_time  # Stop the timer
19 print("Count:", count, "  Elapsed time:", elapsed, "sec")
```

Time function : sleep example

- The sleep function suspends the program's execution for a specified number of seconds.
- The sleep function is useful for controlling the speed of graphical animations.

Listing 6.9: countdown.py

```
1 from time import sleep
2
3 for count in range(10, -1, -1):  #  Range 10, 9, 8, ..., 0
4     print(count)       #  Display the count
5     sleep(1)           #  Suspend execution for 1 second
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Python Lambda

Lambda functions can take any number of arguments:

Example

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

Summarize argument `a`, `b`, and `c` and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Python Lambda

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Error Handling

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

3.4.1 Syntax Errors

The interpreter is designed to execute all valid Python programs. The interpreter reads the Python source code and translates it into executable machine code. This is the translation phase.

If the interpreter detects an invalid program during the translation phase, it will terminate the program's execution and report an error. Such errors result from the programmer's misuse of the language.

A syntax error is a common error that the interpreter can detect when attempting to translate a Python statement into machine language. For example, in English one can say

The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the Python statement

x = y + 2

is syntactically correct because it obeys the rules for the structure of an assignment statement described in Section 2.2. However, consider replacing this assignment statement with a slightly modified version:

y + 2 = x

If a statement like this one appears in a program, the interpreter will issue an error message;

```
>>> y + 2 = x
  File "error.py", line 12
SyntaxError: can't assign to operator
```

3.4.2 Run-time Errors

A syntactically correct Python program still can have problems. Some language errors depend on the context of the program's execution. Such errors are called run-time errors or exceptions. Run-time errors arise after the interpreter's translation phase and during its execution phase.

The interpreter may issue an error for a syntactically correct statement like

x = y + 2

if the variable y has yet to be assigned; for example, if the statement appears at line 12 and by that point y has not been assigned, we are informed:

```
>>> x = y + 2
Traceback (most recent call last):
  File "error.py", line 12, in <module>
NameError: name 'y' is not defined
```

Consider Listing (dividedanger.py) which contains an error that manifests itself only in one particular situation.

Listing 3.4: dividedanger.py

```
1  #  File dividedanger.py
2
3  #  Get two integers from the user
4  dividend, divisor = eval(input('Please enter two numbers to divide: '))
5  #  Divide them and report the result
6  print(dividend, '/', divisor, "=", dividend/divisor)
```

The expression

dividend/divisor

is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely

```
Please enter two integers to divide: 32, 4
32 / 4 = 8.0
```

If the user instead types the numbers 32 and 0, the program reports an error and terminates:

```
Please enter two numbers to divide: 32, 0
Traceback (most recent call last):
  File "C:\Users\rick\Desktop\changeable.py", line 6, in <module>
    print(dividend, '/', divisor, "=", dividend/divisor)
ZeroDivisionError: division by zero
```

Classes and Objects

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

OUTPUT

```
<class '__main__.MyClass'>
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

OUTPUT

```
5
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

OUTPUT

```
John
36
```

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

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

    def myfunc(self):
      print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc(
```

OUTPUT

```
Hello my name is John
```

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

LIST OF OBJECT IN PYTHON CLASS

We can create list of object in Python by appending class **instances** to **list**. By this, every index in the list can point to instance attribute and methods of class and can access them. If you observe it closely, list of objects behaves like an array of structures in C. Let's try to understand it better with help of examples.

```
# Python3 code here creating class
class geeks:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

# creating list
list = []

# appending instances to list
list.append( geeks('Akash', 2) )
list.append( geeks('Deependra', 40) )
list.append( geeks('Reaper', 44) )

for obj in list:
    print( obj.name, obj.roll, sep =' ' )

# We can also access instances attributes
# as list[0].name, list[0].roll and so on.
```

OUTPUT

```
Akash 2
Deependra 40
Reaper 44
```

B.S. Abdur Rahman
Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

Inheritance & Polymorphism

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
class Student(Person):
  pass
```

Note: Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Example

Add a property called `graduationyear` to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year `2019` should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a `year` parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example

Add a method called `welcome` to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Polymorphism in Python

- Polymorphism means multiple forms.
- In python we can find the same operator or function taking multiple forms.
- It also useful in creating different classes which will have class methods with same name.
- That helps in re using a lot of code and decreases code complexity.

Polymorphism in operators

The + operator can take two inputs and give us the result depending on what the inputs are. In the below examples we can see how the integer inputs yield an integer and if one of the input is float then the result becomes a float. Also for strings, they simply get concatenated. This happens automatically because of the way the + operator is created in python.

Polymorphism in operators: Example

Example

```
a = 23
b = 11
c = 9.5
s1 = "Hello"
s2 = "There!"
print(a + b)
print(type(a + b))
print(b + c)
print(type (b + c))
print(s1 + s2)
print(type(s1 + s2))
```

Running the above code gives us the following result –

Output

```
34
20.5
HelloThere!
```

Polymorphism in in-built functions

We can also see that different python functions can take inputs of different types and then process them differently. When we supply a string value to len() it counts every letter in it. But if we five tuple or a dictionary as an input, it processes them differently.

Example

```
str = 'Hi There !'
tup = ('Mon','Tue','wed','Thu','Fri')
lst = ['Jan','Feb','Mar','Apr']
dict = {'1D':'Line','2D':'Triangle','3D':'Sphere'}
print(len(str))
print(len(tup))
print(len(lst))
print(len(dict))
```

Running the above code gives us the following result –

Output

```
10
5
4
3
```

Polymorphism in user-defined methods

We can create methods with same name but wrapped under different class names. So we can keep calling the same method with different class name pre-fixed to get different result. In the below example we have two classes, rectangle and circle to get their perimeter and area using same methods.

Example

```
from math import pi

class Rectangle:
   def __init__(self, length, breadth):
      self.l = length
      self.b = breadth
   def perimeter(self):
      return 2*(self.l + self.b)
   def area(self):
      return self.l * self.b
```

```
class Circle:
    def __init__(self, radius):
        self.r = radius
    def perimeter(self):
        return 2 * pi * self.r
    def area(self):
        return pi * self.r ** 2

# Initialize the classes
rec = Rectangle(5,3)
cr = Circle(4)
print("Perimter of rectangel: ",rec.perimeter())
print("Area of rectangel: ",rec.area())

print("Perimter of Circle: ",cr.perimeter())
print("Area of Circle: ",cr.area())
```

Running the above code gives us the following result –

Output

```
Perimter of rectangel: 16
Area of rectangel: 15
Perimter of Circle: 25.132741228718345
Area of Circle: 50.26548245743669
```

Exception Handling

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*

EXCEPTION HANDLING

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks –

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Thank You

Course Instructor : *Dr.A.Abdul Azeez Khan & Dr. K.Javubar Sathick*