

[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

**Standard Library**

Built-in Functions

Built-in functions are Readily available for reuse.

Some of the built Functions are

1. print()
2. max()
3. min()
4. len()

and many more..

Standard Library

Python provides several such useful values (constants), classes and functions.

This collection of predefined utilities is referred as the **Python Standard Library**

All these functionalities are organized into different modules.

* In Python context, any file containing a Python code is called a **module**
* These modules are further organized into folders known as **packages**

Different modules are:

1. collections
2. random
3. datetime
4. math and many more..

Working with Standard Library

To use a functionality defined in a module we need to import that module in our program.

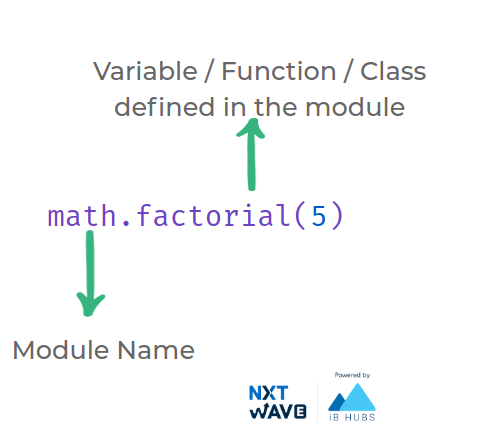


1

import module\_name

PYTHON

Math Module

**math** module provides us to access some common *math functions* and *constants*. 

**Code**



1

2

3

import math

print(math.factorial(5))

print(math.pi)

PYTHON

**Output**



120

3.141592653589793

Importing module

Importing a module and giving it a new name (aliasing)

**Code**



1

2

import math as m1

print(m1.factorial(5))

PYTHON

**Output**



120

Importing from a Module

We can import just a specific definition from a module.

**Code**



1

2

from math import factorial

print(factorial(5))

PYTHON

**Output**



120

Aliasing Imports

We can also import a specific definition from a module and alias it

**Code**



1

2

from math import factorial as fact

print(fact(5))

PYTHON

**Output**



120

Random module

Randomness is useful in whenever uncertainty is required.

*For example*: Rolling a dice, flipping a coin, etc.,

random

module provides us utilities to create randomness.



Randint

randint()

is a function in random module which returns a random integer in the given interval.

**Code**



1

2

3

import random

random\_integer = random.randint(1, 10)

print(random\_integer)

PYTHON

**Output**



8

Choice

choice()

is a function in random module which returns a random element from the sequence.

**Code**



1

2

3

import random

random\_ele = random.choice(["A","B","C"])

print(random\_ele)

PYTHON

**Output**



B

To know more about **Python Standard Library**, go through the authentic python documentation - https://docs.python.org/3/library/

**Map Filter and Reduce**

We worked with different sequences (list, tuples, etc.)

To simplify working with sequences we can use

map()

,

filter()

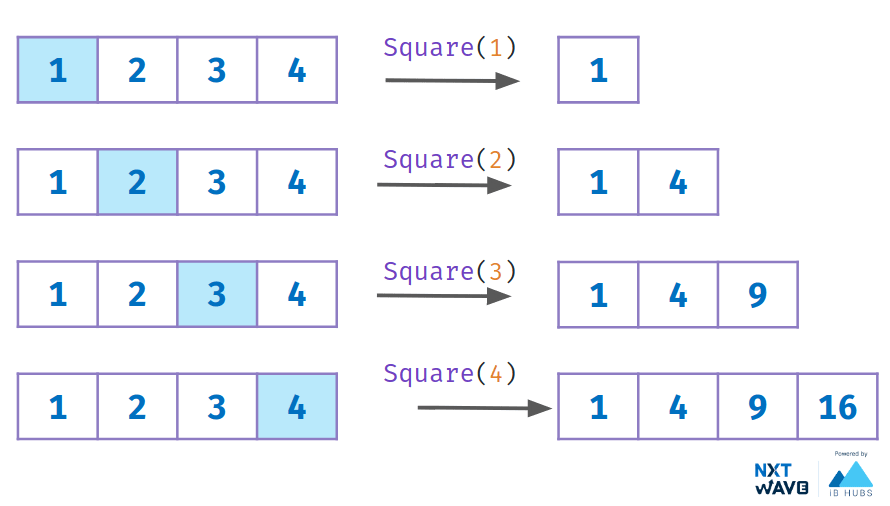
and

reduce()

functions.

Map

map()

applies a given function to each item of a sequence (list, tuple etc.) and returns a sequence of the results. 

*Example - 1*

**Code**



1

2

3

4

5

6

def square(n):

return n \* n

numbers = [1, 2, 3, 4]

result = map(square, numbers)

numbers\_square = list(result)

print(numbers\_square)

PYTHON

**Output**



[1, 4, 9, 16]

*Example - 2*

**Code**



1

2

numbers = list(map(int, input().split()))

print(numbers)

PYTHON

**Input**



1 2 3 4

**Output**

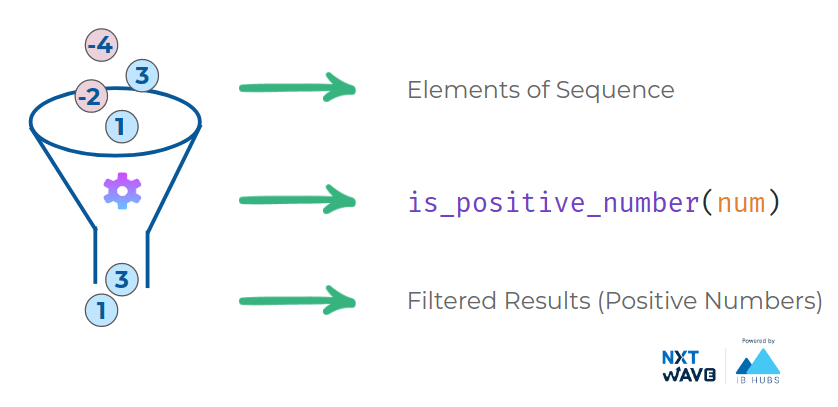


[1, 2, 3, 4]

Filter

filter()

method filters the elements of a given sequence based on the result of given function.

The function should return True/False 

**Code**



1

2

3

4

5

6

def is\_positive\_number(num):

return num > 0

list\_a = [1, -2, 3, -4]

positive\_nums = filter(is\_positive\_number, list\_a)

print(list(positive\_nums))

PYTHON

**Output**

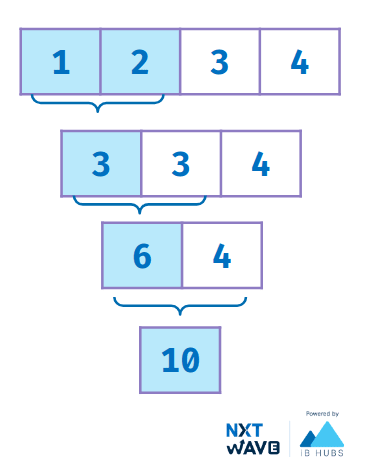


[1, 3]

Reduce

reduce()

function is defined in the functools module.



**Code**



1

2

3

4

5

6

7

8

from functools import reduce

def sum\_of\_num(a, b):

return a+b

list\_a = [1, 2, 3, 4]

sum\_of\_list = reduce(sum\_of\_num, list\_a)

print(sum\_of\_list)

PYTHON

**Output**



10

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

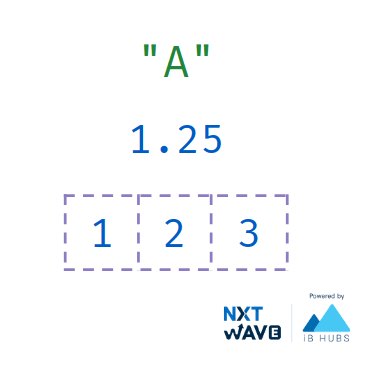
**Cheat Sheet**

Cheat Sheet

**Scope & Namespaces**

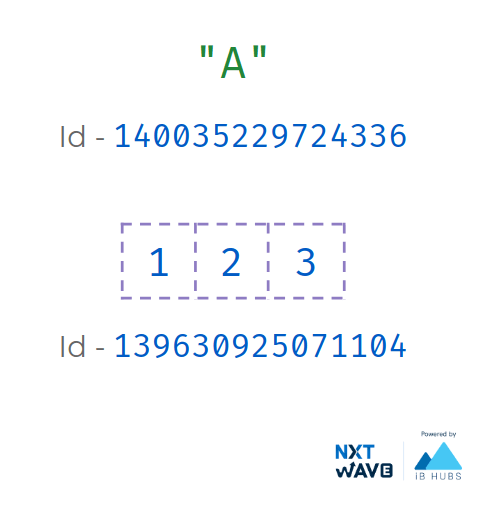
Object

In general, anything that can be assigned to a variable in Python is referred to as an **object**.

Strings, Integers, Floats, Lists, Functions, Module etc. are all objects. 

Identity of an Object

Whenever an object is created in Python, it will be given a **unique identifier (id)**.This unique id can be different for each time you run the program.

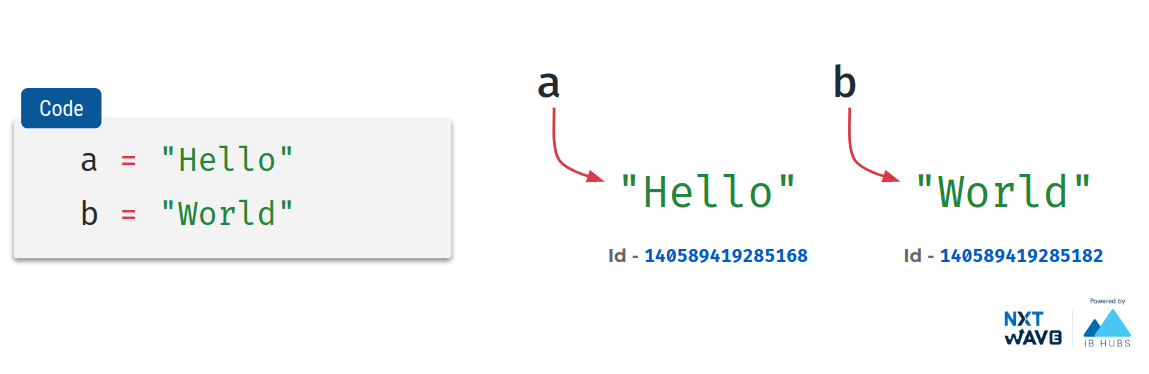


Every object that you use in a Python Program will be stored in Computer Memory

The unique id will be related to the location where the object is stored in the **Computer Memory**.

Name of an Object

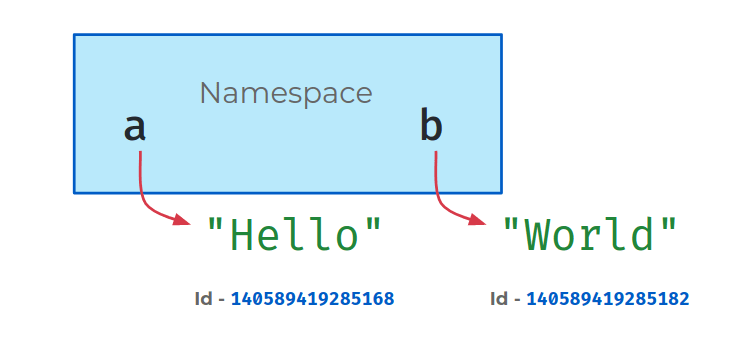
**Name** or **Identifier** is simply a name given to an object.

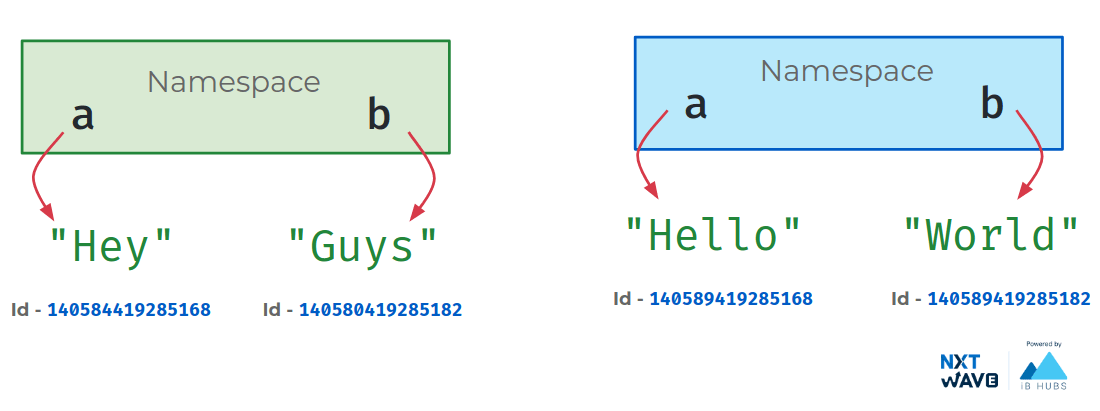


Namespaces

A **namespace** is a collection of currently defined names along with information about the object that the name references.

It ensures that names are **unique** and won’t lead to any conflict.

Namespaces allow us to have the same name referring different things in **different namespaces**.



**Code**



1

2

3

4

5

6

7

8

9

10

11

def greet\_1():

a = "Hello"

print(a)

print(id(a))

def greet\_2():

a = "Hey"

print(a)

print(id(a))

print("Namespace - 1")

PYTHON

**Output**



Namespace - 1

Hello

140639382368176

Namespace - 2

Hey

140639382570608

Types of namespaces

As Python executes a program, it creates namespaces as necessary and forgets them when they are no longer needed.

Different namespaces are:

1. Built-in
2. Global
3. Local

Built-in Namespace

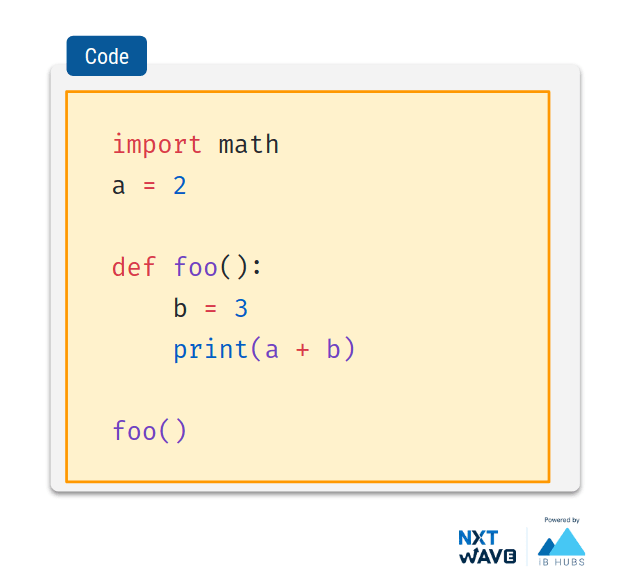
Created when we start executing a Python program and exists as long as the program is running.

This is the reason that built-in functions like **id(), print()** etc. are always available to us from any part of the program.

Global Namespace

This namespace includes all names defined directly in a module (outside of all functions).

It is created when the module is loaded, and it lasts until the program ends.



Local Namespace

Modules can have various

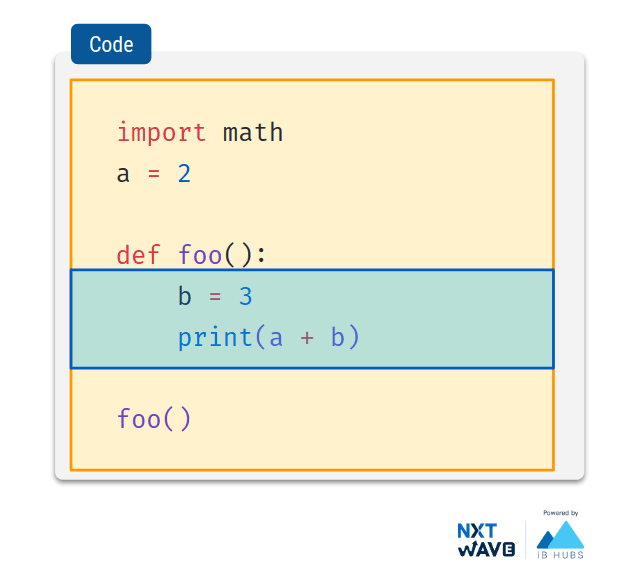
functions

and

classes

.

A new local namespace is created when a function is called, which lasts until the function returns.



Scope of a Name

The scope of a name is the region of a program in which that name has meaning.

Python searches for a name from the inside out, looking in the

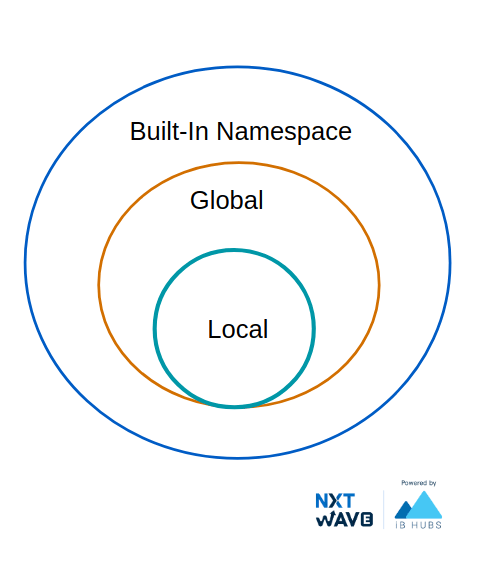
local

,

global

, and

finally

the built-in namespaces. 

Global variables

In Python, a variable defined outside of all functions is known as a **global variable**.

This variable name will be part of **Global Namespace**.

*Example 1*

**Code**



1

2

3

4

5

6

7

x = "Global Variable"

print(x)

def foo():

print(x)

foo()

PYTHON

**Output**



Global Variable

Global Variable

*Example 2*

**Code**



1

2

3

4

5

6

def foo():

print(x)

x = "Global Variable"

foo()

PYTHON

**Output**

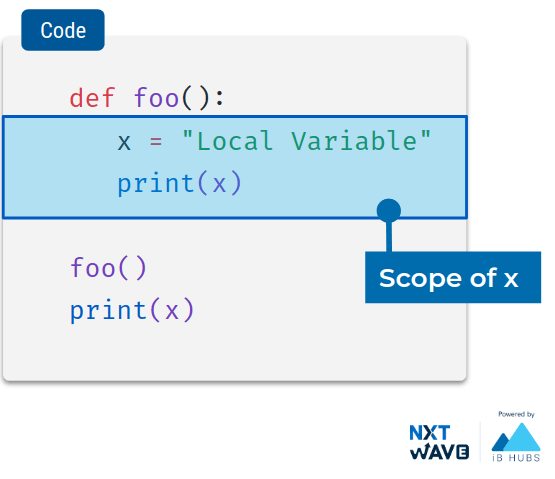


Global Variable

Local Variables

In Python, a variable defined inside a function is a local variable.

This variable name will be part of the Local Namespace which will be created when the function is called and lasts until the function returns.



**Code**



1

2

3

4

5

6

def foo():

x = "Local Variable"

print(x)

foo()

print(x)

PYTHON

**Output**



Local Variable

NameError: name 'x' is not defined

As,

x

is not declared before assignment, python throws an error.

Local Import

**Code**



1

2

3

4

5

6

def foo():

import math

print(math.pi)

foo()

print(math.pi)

PYTHON

**Output**



3.141592653589793

NameError: name 'math' is not defined

Local Variables & Global Variables

**Code**



1

2

3

4

5

6

7

8

9

x = "Global Variable"

def foo():

x = "Local Variable"

print(x)

print(x)

foo()

print(x)

PYTHON

**Output**



Global Variable

Local Variable

Global Variable

Modifying Global Variables

global

keyword is used to define a name to refer to the value in Global Namespace.

**Code**



1

2

3

4

5

6

7

8

9

10

x = "Global Variable"

def foo():

global x

x = "Global Change"

print(x)

print(x)

foo()

print(x)

PYTHON

**Output**



Global Variable

Global Change

Global Change

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

**Errors & Exceptions**

There are two major kinds of errors:

1. Syntax Errors
2. Exceptions

Syntax Errors

Syntax errors are parsing errors which occur when the code is not adhering to **Python Syntax**.

**Code**



1

if True print("Hello")

PYTHON

**Output**



SyntaxError: invalid syntax

When there is a syntax error, the program will **not** execute even if that part of code is not used.

**Code**



1

2

3

4

print("Hello")

def greet():

print("World"

PYTHON

**Output**



SyntaxError: unexpected EOF while parsing

Notice that in the above code, the syntax error is inside the

greet

function, which is not used in rest of the code.

Exceptions

Even when a statement or expression is **syntactically correct**, it may cause an **error** when an attempt is made to execute it.

Errors detected during execution are called **exceptions**.

Example Scenario

We wrote a program to download a Video over the Internet.

* Internet is disconnected during the download
* We do not have space left on the device to download the video

*Example 1*

Division Example

Input given by the user is not within expected values.

**Code**



1

2

3

4

def divide(a, b):

return a / b

divide(5, 0)

PYTHON

**Output**



ZeroDivisionError: division by zero

*Example 2*

Input given by the user is not within expected values.

**Code**



1

2

3

4

5

def divide(a, b):

return a / b

divide("5", "10")

PYTHON

**Output**



TypeError: unsupported operand type(s) for /: 'str' and 'str'

*Example 3*

Consider the following code, which is used to update the quantity of items in store.

**Code**



1

2

3

4

5

6

7

8

9

10

class Store:

def \_\_init\_\_(self):

self.items = {

"milk" : 20, "bread" : 30, }

def add\_item(self, name, quantity):

self.items[name] += quantity

s = Store()

s.add\_item('biscuits', 10)

PYTHON

**Output**



KeyError: 'biscuits'

Working With Exceptions

What happens when your code runs into an exception during execution?

**The application/program crashes.**

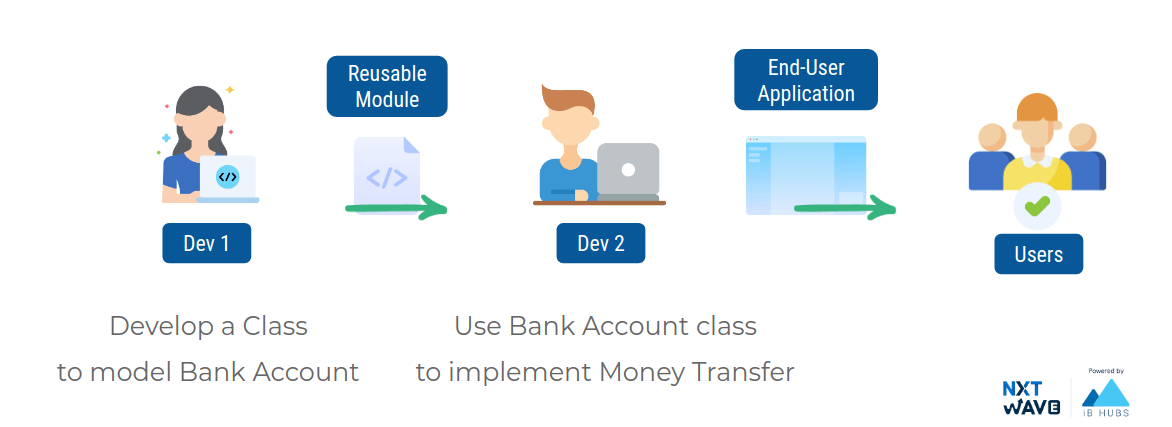
End-User Applications

When you develop applications that are directly used by end-users, you need to **handle different possible exceptions** in your code so that the application will not crash.

Reusable Modules

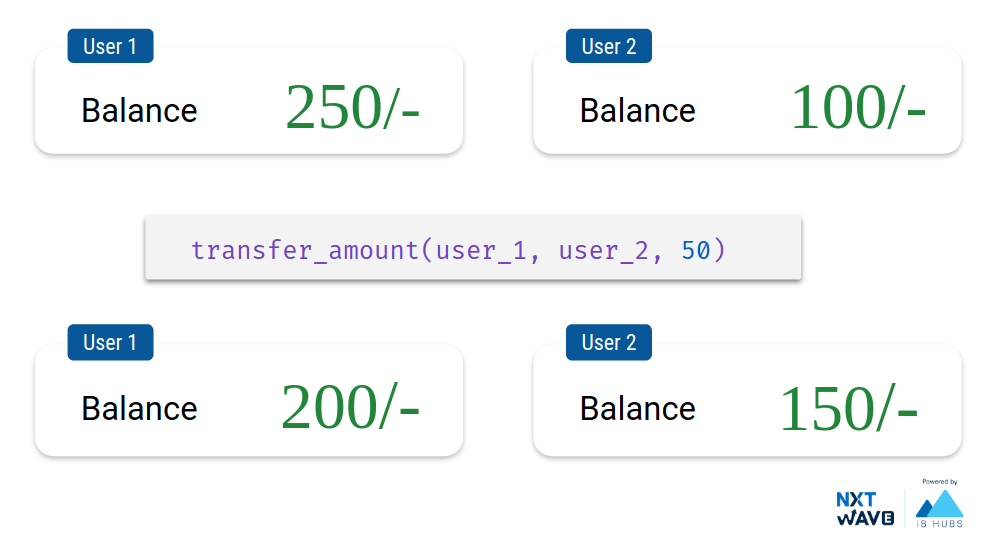
When you develop modules that are used by other developers, you should **raise exceptions** for different scenarios so that other developers can handle them.

Money Transfer App Scenario

Let’s consider we are creating an app that allows users to transfer money between them.  


Bank Account Class

*Example 1*



**Code**



1

2

3

4

5

6

7

8

9

10

11

12

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

PYTHON

**Output**



User 1 Balance: 250/-

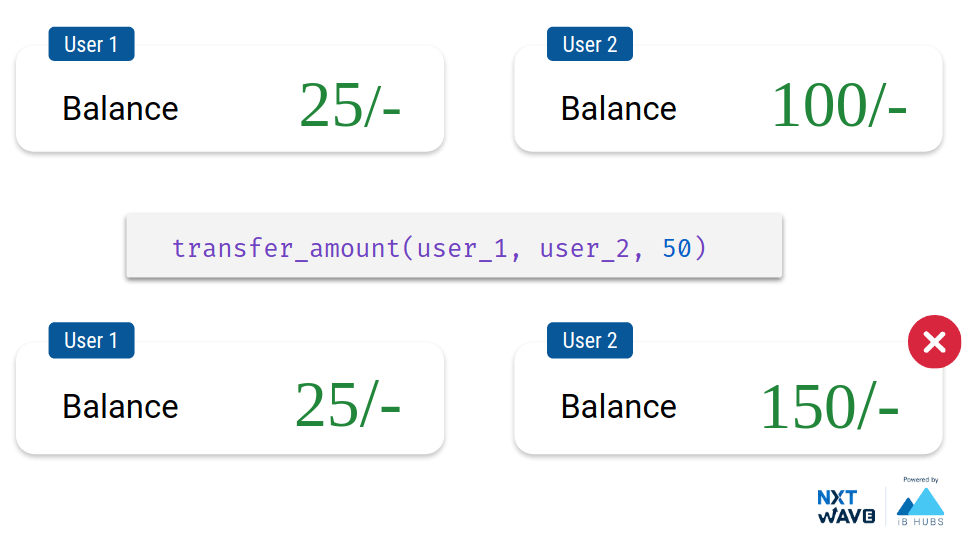
User 2 Balance: 100/-

Transferring 50/- from User 1 to User 2

User 1 Balance: 200/-

User 2 Balance: 150/-

*Example 2*



**Code**



1

2

3

4

5

6

7

8

9

10

11

12

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

PYTHON

**Output**



User 1 Balance: 25/-

User 2 Balance: 100/-

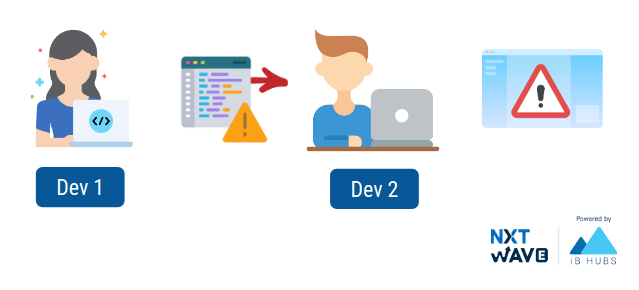
Insufficient Funds

Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

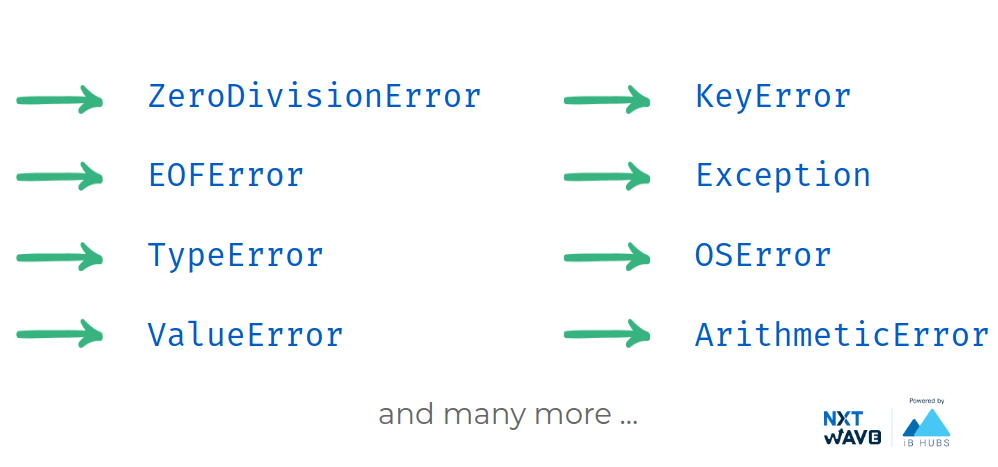
User 2 Balance: 150/-

Raising Exceptions

When your code enters an unexpected state, **raise** an exception to communicate it.  


Built-in Exceptions

Different **exception classes** which are raised in different scenarios.



You can use the built-in exception classes with **raise** keyword to **raise an exception** in the program.

**Code**

We can pass message as **argument** .



1

raise ValueError("Unexpected Value!!")

PYTHON

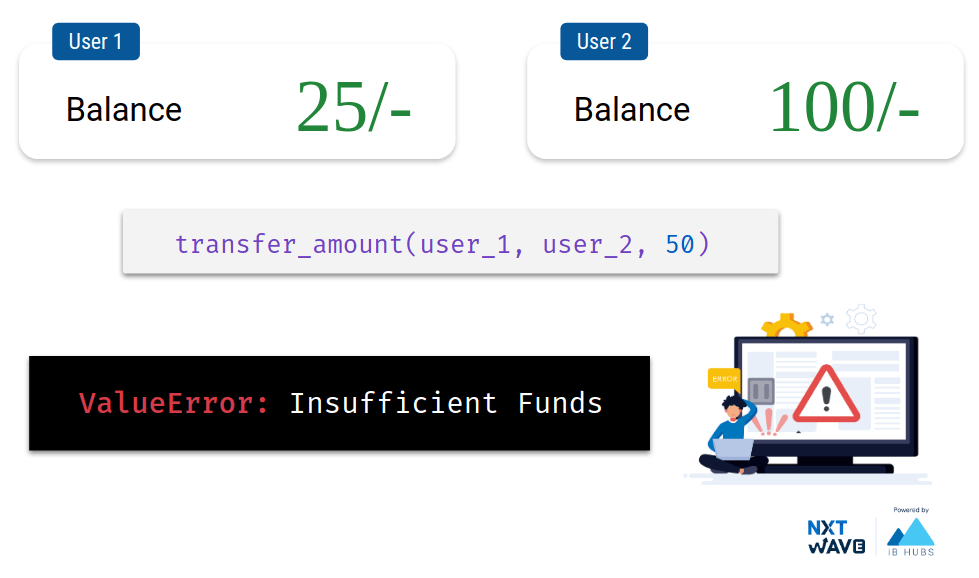
**Output**



ValueError:Unexpected Value!!

Bank Account Class

*Example 1*



**Code**



1

2

3

4

5

6

7

8

9

10

11

12

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

PYTHON

**Output**



User 1 Balance: 25/-

User 2 Balance: 100/-

ValueError: Insufficient Funds

Handling Exceptions

Python provides a way to **catch** the exceptions that were raised so that they can be properly handled.

* Exceptions can be handled with **try-except** block.
* Whenever an exception occurs at some line in try block, the execution stops at that line and jumps to except block.



1

2

3

4

5

6

try:

# Write code that

# might cause exceptions.

except:

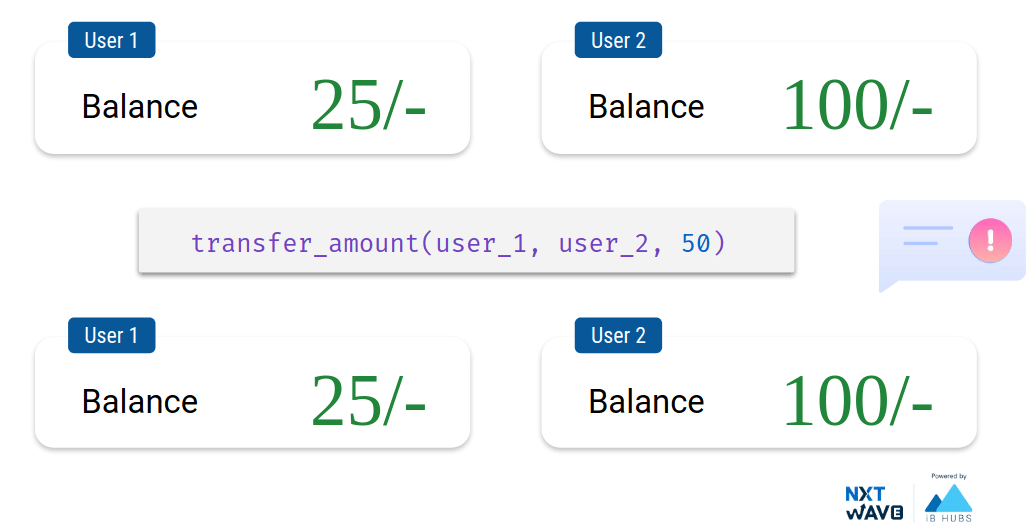
# The code to be run when

# there is an exception.

PYTHON

Transfer Amount

*Example 1*



**Code**



1

2

3

4

5

6

7

8

9

10

11

12

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

PYTHON

**Output**



User 1 Balance: 25/-

User 2 Balance: 100/-

False

Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

User 2 Balance: 100/-

Summary

**Reusable Modules**

* While developing reusable modules, we need to raise Exceptions to stop our code from being used in a bad way.

**End-User Applications**

* While developing end-user applications, we need to handle Exceptions so that application will not crash when used.

Handling Specific Exceptions

We can specifically mention the **name of exception** to catch all exceptions of that specific type.

**Syntax**



1

2

3

4

5

6

try:

# Write code that

# might cause exceptions.

except Exception:

# The code to be run when

# there is an exception.

PYTHON

*Example 1*

**Code**



1

2

3

4

5

6

7

8

9

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except:

print("Unhandled Exception")

PYTHON

**Input**



5

0

**Output**



Denominator can't be 0

*Example 2*

**Code**

Input given by the user is not within expected values.



1

2

3

4

5

6

7

8

9

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except:

print("Unhandled Exception")

PYTHON

**Input**



12

a

**Output**



Unhandled Exception

We can also access the handled exception in an **object**.

**Syntax**



1

2

3

4

5

6

try:

# Write code that

# might cause exceptions.

except Exception as e:

# The code to be run when

# there is an exception.

PYTHON

**Code**



1

2

3

4

5

6

7

8

9

10

11

12

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

PYTHON

**Output**



User 1 Balance: 25/-

User 2 Balance: 100/-

Insufficient Funds

<class 'ValueError'>

('Insufficient Funds',)

False

Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

User 2 Balance: 100/-

Handling Multiple Exceptions

We can write **multiple exception blocks** to handle different types of exceptions differently.

**Syntax**



1

2

3

4

5

6

7

8

9

try:

# Write code that

# might cause exceptions.

except Exception1:

# The code to be run when

# there is an exception.

except Exception2:

# The code to be run when

# there is an exception.

PYTHON

*Example 1*

**Code**



1

2

3

4

5

6

7

8

9

10

11

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except ValueError:

print("Input should be an integer")

except:

print("Something went wrong")

PYTHON

**Input**



5

0

**Output**



Denominator can't be 0

*Example 2*

**Code**



1

2

3

4

5

6

7

8

9

10

11

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except ValueError:

print("Input should be an integer")

except:

print("Something went wrong")

PYTHON

**Input**



12

a

**Output**



Input should be an integer

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1



[Home](https://learning.ccbp.in/)Code Playground

Nakka Vijay Kumar

**NK**

**Cheat Sheet**

Cheat Sheet

**Working With Dates & Times**

Datetime

Python has a built-in **datetime** module which provides convenient objects to work with dates and times.

**Code**



1

import datetime

PYTHON

Datetime classes

Commonly used **classes** in the datetime module are:

* date class
* time class
* datetime class
* timedelta class

Working with 'date' class

Representing Date

A date object can be used to represent any valid **date** (year, month and day).

**Code**



1

2

3

4

import datetime

date\_object = datetime.date(2019, 4, 13)

print(date\_object)

PYTHON

**Output**



2019-04-13

Date Object

**Code**



1

2

3

from datetime import date

date\_obj = date(2022, 2, 31)

print(date\_obj)

PYTHON

**Output**



ValueError: day is out of range for month

Today’s Date

Class method

today()

returns a date object with **today’s date**.

**Code**



1

2

3

4

import datetime

date\_object = datetime.date.today()

print(date\_object)

PYTHON

**Output**



2021-02-05

Attributes of Date Object

**Code**



1

2

3

4

5

6

from datetime import date

date\_object = date(2019, 4, 13)

print(date\_object.year)

print(date\_object.month)

print(date\_object.day)

PYTHON

**Output**



2019

4

13

Working with ‘time’ Class

Representing Time

A time object can be used to represent any valid **time** (hours, minutes and seconds).

**Code**



1

2

3

4

from datetime import time

time\_object = time(11, 34, 56)

print(time\_object)

PYTHON

**Output**



11:34:56

Attributes of Time Object

**Code**



1

2

3

4

5

6

7

from datetime import time

time\_object = time(11, 34, 56)

print(time\_object)

print(time\_object.hour)

print(time\_object.minute)

print(time\_object.second)

PYTHON

**Output**



11:34:56

11

34

56

Working with ‘datetime’ Class

Datetime

The datetime class represents a valid **date and time** together.

*Example - 1*

**Code**



1

2

3

4

5

6

7

from datetime import datetime

date\_time\_obj = datetime(2018, 11, 28, 10, 15, 26)

print(date\_time\_obj.year)

print(date\_time\_obj.month)

print(date\_time\_obj.hour)

print(date\_time\_obj.minute)

PYTHON

**Output**



2018

11

10

15

*Example - 2* It gives the current date and time

**Code**



1

2

3

4

import datetime

datetime\_object = datetime.datetime.now()

print(datetime\_object)

PYTHON

**Output**



2021-02-05 09:26:08.077473

DateTime object

**Code**



1

2

3

from datetime import datetime

date\_time\_obj = datetime(2018, 11, 28)

print(date\_time\_obj)

PYTHON

**Output**



2018-11-28 00:00:00

Formatting Datetime

The datetime classes have

strftime(format)

method to format the datetime into any required format like

* mm/dd/yyyy
* dd-mm-yyyy

| Format Specifier | Meaning | Example |
| --- | --- | --- |
| %y | Year without century as a zero-padded decimal number | 19, 20, ... |
| %Y | Year with century as a decimal number | 2019, 2020, ... |
| %b | Month as abbreviated name | Jan, Feb, ... |
| %B | Month as full name | January, February |
| %m | Month as a zero-padded decimal number | 01, 02, …, 12 |
| %d | Day of the month as a zero-padded decimal number | 01, 02, …, 31 |
| %a | Weekday as abbreviated name | Sun, Mon, ... |
| %A | Weekday as full name | Sunday, Monday, ... |
| %H | Hour (24-hour clock) as a zero-padded decimal number | 00, 01, …, 23 |
| %I | Hour (12-hour clock) as a zero-padded decimal number | 01, 02, …, 12 |
| %p | AM or PM | AM, PM |
| %M | Minute as a zero-padded decimal number | 00, 01, …, 59 |
| %S | Second as a zero-padded decimal number | 00, 01, …, 59 |

**Code**



1

2

3

4

5

6

7

8

from datetime import datetime

now = datetime.now()

formatted\_datetime\_1 = now.strftime("%d %b %Y %I:%M:%S %p")

print(formatted\_datetime\_1)

formatted\_datetime\_2 = now.strftime("%d/%m/%Y, %H:%M:%S")

print(formatted\_datetime\_2)

PYTHON

**Output**



05 Feb 2021 09:26:50 AM

05/02/2021, 09:26:50

Parsing Datetime

The class method

strptime()

creates a **datetime object** from a given string representing date and time.

**Code**



1

2

3

4

5

6

7

from datetime import datetime

date\_string = "28 November, 2018"

print(date\_string)

date\_object = datetime.strptime(date\_string, "%d %B, %Y")

print(date\_object)

PYTHON

**Output**



28 November, 2018

2018-11-28 00:00:00

Working with ‘timedelta’ Class

Timedelta object represents **duration**.

*Example 1*

**Code**



1

2

3

4

from datetime import timedelta

delta = timedelta(days=365, hours=4)

print(delta)

PYTHON

**Output**



365 days, 4:00:00

*Example 2*

**Code**



1

2

3

4

5

6

from datetime import timedelta, datetime

delta = timedelta(days=365)

current\_datetime = datetime.now()

print(current\_datetime)

next\_year\_datetime = current\_datetime + delta

print(next\_year\_datetime)

PYTHON

**Output**



2021-02-05 09:28:30.239095

2022-02-05 09:28:30.239095

Calculating Time Difference

**Code**



1

2

3

4

5

6

7

import datetime

dt1 = datetime.datetime(2021, 2, 5)

dt2 = datetime.datetime(2022, 1, 1)

duration = dt2 - dt1

print(duration)

print(type(duration))

PYTHON

**Output**



330 days, 0:00:00

<class 'datetime.timedelta'>

MARK AS COMPLETED

Notes

Discussions

Notes

NEW NOTE

CANCELSAVE

[**Download App**](https://play.google.com/store/apps/details?id=co.ibhubs.chat)

**Personal Manager**



1