

## Cheat Sheet

### Intro to Programming with Python

#### Software

Software is a set of instructions to the hardware.

#### Programming

Programming means writing the instructions to create a software.

#### Code

The instructions that we write to create software is called **Code**.

#### Syntax

Similar to Grammar rules in English, Hindi, each programming language has a unique set of rules. These rules are called the **Syntax** of a Programming Language.

### Why Python

Python is an easy to learn, powerful programming language. With Python, it is possible to create programs with minimal amount of code. Look at the code in Java and Python used for printing the message "**Hello World**"

#### Java:

```
1  class Main {  
2      public static void main(String[] args) {  
3          System.out.println("Hello World");  
4      }  
5  }
```

JAVA

## Python:

PYTHON

```
1 print("Hello World")
```

### Applications of Python

Python is a versatile language which has applications in almost every field

- Artificial intelligence (AI)
- Machine Learning (ML)
- Big Data
- Smart Devices/Internet of Things (IoT)
- Cyber Security
- Game Development
- Backend Development, etc.

### Career Opportunities

Python developers have plenty of opportunities across the world

- DevOps Engineer
- Software Developer
- Data Analyst
- Data Scientist
- Machine Learning (ML) Engineer
- AI Scientist, etc.

### Hello World Program in Python

Here is a simple Python code that you can use to display the message "**Hello World**"

#### Code

PYTHON

```
1 print("Hello World!")
```

## Output

```
Hello World!
```

## Possible Mistakes

Possible mistakes you may make while writing Python code to display the message "Hello World"

- Spelling Mistake in print

```
1 prnt("Hello World!")
```

PYTHON

- Uppercase 'P' in Print

```
1 Print("Hello World!")
```

PYTHON

- Missing quotes

```
1 print(Hello World!)
```

PYTHON

- Missing parentheses

```
1 print("Hello World!"
```

PYTHON

## Printing Without Quotes

If we want to print the numerical result of  $2 + 5$ , we do not add quotes.

## Code

```
1 print(2 + 5)
```

PYTHON

## Output

```
7
```

If we want to print the exact message "2 + 5", then we add the quotes.

## Code

```
1 print("2 + 5")
```

PYTHON

## Output

```
2 + 5
```

## Calculations with python

### Addition

Addition is denoted by

+ sign. It gives the sum of two numbers.

## Code

```
1 print(2 + 5)
2 print(1 + 1.5)
```

PYTHON

## Output

7

2.5

## Subtraction

Subtraction is denoted by

- sign. It gives the difference between two numbers.

Code

PYTHON

```
1 print(5 - 2)
```

Output

3

## Multiplication

Multiplication is denoted by

- \* sign.

Code

PYTHON

```
1 print(2 * 5)
2 print(5 * 0.5)
```

Output

10

10

2.5

## Division

Division is denoted by

/ sign.

### Code

PYTHON

```
1 print(5 / 2)
2 print(4/2)
```

### Output

2.5

2.0



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

# Variables and Data Types

## Variables

Variables are like containers for storing values.

Values in the variables can be changed.

## Values

Consider that variables are like containers for storing information.

In context of programming, this information is often referred to as value.

## Data Type

In programming languages, every value or data has an associated type to it known as data type.

Some commonly used data types

- String
- Integer
- Float
- Boolean

This data type determines how the value or data can be used in the program. For example, mathematical operations can be done on Integer and Float types of data.

## String

A String is a stream of characters enclosed within quotes.

Stream of Characters

- Capital Letters ( A – Z )
- Small Letters ( a – z )
- Digits ( 0 – 9 )

- Special Characters (~ ! @ # \$ % ^ . ? ,)
- Space

## Some Examples

- "Hello World!"
- "some@example.com"
- "1234"

## Integer

All whole numbers (positive, negative and zero) without any fractional part come under Integers.

### Examples

...-3, -2, -1, 0, 1, 2, 3,...

## Float

Any number with a decimal point

24.3, 345.210, -321.86

## Boolean

In a general sense, anything that can take one of two possible values is considered a Boolean. Examples include the data that can take values like

- True or False
- Yes or No
- 0 or 1
- On or Off , etc.

As per the Python Syntax,

True and False are considered as Boolean values. Notice that both start with a capital letter.

### Assigning Value to Variable

The following is the syntax for assigning an integer value

10 to a variable age

```
1 age = 10
```

PYTHON

Here the equals to

= sign is called as **Assignment Operator** as it is used to assign values to variables.



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Sequence of Instructions

#### Program

A program is a sequence of instructions given to a computer.

#### Defining a Variable

A variable gets created when you assign a value to it for the first time.

#### Code

```
1 age = 10
```

PYTHON

#### Printing Value in a Variable

#### Code

```
1 age = 10
2 print(age)
```

PYTHON

#### Output

```
10
```

#### Code

PYTHON

```
1 age = 10  
2 print("age")
```

## Output

```
age
```

Variable name enclosed in quotes will print variable rather than the value in it.

If you intend to print value, do not enclose the variable in quotes.

## Order of Instructions

Python executes the code line-by-line.

## Code

PYTHON

```
1 print(age)  
2 age = 10
```

## Output

```
NameError: name 'age' is not defined
```

## Variable

age is not created by the time we tried to print.

## Spacing in Python

Having spaces at the beginning of line causes errors.

## Code

PYTHON

```
1 a = 10 * 5
2 b = 5 * 0.5
3 b = a + b
```

## Output

```
File "main.py", line 3
    b = a + b
    ^
IndentationError: unexpected indent
```

## Variable Assignment

Values in the variables can be changed.

## Code

PYTHON

```
1 a = 1
2 print(a)
3 a = 2
4 print(a)
```

## Output

```
1
2
```

## Examples of Variable Assignment

## Code

PYTHON

```
1 a = 2
2 print(a)
3 a = a + 1
4 print(a)
```

## Output

```
2
3
```

## Code

PYTHON

```
1 a = 1
2 b = 2
3 a = b + 1
4 print(a)
5 print(b)
```

## Output

```
3
2
```

## Expression

An expression is a valid combination of values, variables and operators.

### Examples

a \* b

```
a + 2  
5 * 2 + 3 * 4
```

## BODMAS

The standard order of evaluating an expression

- *Brackets* (B)
- *Orders* (O)
- *Division* (D)
- *Multiplication* (M)
- *Addition* (A)
- *Subtraction* (S)

### Step by Step Explanation

```
(5 * 2) + (3 * 4)  
(10) + (12)  
22
```

### Code

PYTHON

```
1 print(10 / 2 + 3)  
2 print(10 / (2 + 3))
```

### Output

```
8.0  
2.0
```



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Inputs and Outputs Basics

#### Take Input From User

`input()` allows flexibility to take the input from the user. Reads a line of input as a string.

#### Code

PYTHON

```
1 username = input()  
2 print(username)
```

#### Input

Ajay

#### Output

Ajay

### Working with Strings

#### String Concatenation

Joining strings together is called string concatenation.

## Code

PYTHON

```
1 a = "Hello" + " " + "World"  
2 print(a)
```

## Output

```
Hello World
```

## Concatenation Errors

String Concatenation is possible only with strings.

## Code

PYTHON

```
1 a = "*" + 10  
2 print(a)
```

## Output

```
File "main.py", line 1  
  a = "*" + 10  
          ^  
TypeError:  
can only concatenate str (not "int") to str
```

## String Repetition

\* operator is used for repeating strings any number of times as required.

## Code

PYTHON

```
1 a = "*" * 10
2 print(a)
```

## Output

```
*****
```

## Code

PYTHON

```
1 s = "Python"
2 s = ("* " * 3) + s + (" *" * 3)
3 print(s)
```

## Output

```
* * * Python * * *
```

## Length of String

`len()` returns the number of characters in a given string.

## Code

PYTHON

```
1 username = input()
2 length = len(username)
3 print(length)
```

## Input

```
Ravi
```

## Output

```
4
```

## String Indexing

We can access an individual character in a string using their positions (which start from 0) .

These positions are also called as index.

## Code

PYTHON

```
1 username = "Ravi"  
2 first_letter = username[0]  
3 print(first_letter)
```

## Output

```
R
```

## IndexError

Attempting to use an index that is too large will result in an error:

## Code

PYTHON

```
1 username = "Ravi"  
2 print(username[4])
```

### Output

```
IndexError: string index out of range
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Type Conversion

#### String Slicing

Obtaining a part of a string is called string slicing.

#### Code

PYTHON

```
1 variable_name[start_index:end_index]
```

- Start from the `start_index` and stops at `end_index`
- `end_index` is not included in the slice.

#### Code

PYTHON

```
1 message = "Hi Ravi"
2 part = message[3:7]
3 print(part)
```

#### Output

Ravi

#### Slicing to End

If end index is not specified, slicing stops at the end of the string.

## Code

PYTHON

```
1 message = "Hi Ravi"
2 part = message[3:]
3 print(part)
```

## Output

```
Ravi
```

## Slicing from Start

If start index is not specified, slicing starts from the index 0.

## Code

PYTHON

```
1 message = "Hi Ravi"
2 part = message[:2]
3 print(part)
```

## Output

```
Hi
```

## Checking Data Type

Check the datatype of the variable or value using

```
type()
```

## Printing Data Type

### Code

PYTHON

```
1 print(type(10))
2 print(type(4.2))
3 print(type("Hi"))
```

### Output

```
<class 'int'>
<class 'float'>
<class 'str'>
```

## Type Conversion

Converting the value of one data type to another data type is called *Type Conversion* or *Type Casting*.

We can convert

- String to Integer
- Integer to Float
- Float to String and so on.

### String to Integer

`int()` converts valid data of any type to integer

### Code

PYTHON

```
1 a = "5"
2 a = int(a)
3 print(type(a))
4 print(a)
```

## Output

```
<class 'int'>  
5
```

## Invalid Integer Conversion

### Code

PYTHON

```
1 a = "Five"  
2 a = int(a)  
3 print(type(a))
```

### Output

```
ValueError:  
invalid literal for int() with base 10: 'Five'
```

### Code

PYTHON

```
1 a = "5.0"  
2 a = int(a)  
3 print(type(a))
```

### Output

```
invalid literal for int() with base 10: '5.0'
```

## Adding Two Numbers

## Code

PYTHON

```
1 a = input()
2 a = int(a)
3 b = input()
4 b = int(b)
5 result = a + b
6 print(result)
```

## Input

```
2
3
```

## Output

```
5
```

## Integer to String

`str()` converts data of any type to a string.

## Code

PYTHON

```
1 a = input()
2 a = int(a)
3 b = input()
4 b = int(b)
5 result = a + b
6 print("Sum: " + str(result))
```

### Input

2

3

### Output

Sum: 5

### Summary

1. int() -> Converts to integer data type
2. float() -> Converts to float data type
3. str() -> Converts to string data type
4. bool() -> Converts to boolean data type



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Relational Operators

Relational Operators are used to compare values.

Gives

True or False as the result of a comparison.

These are different relational operators

Operator	Name
>	Is greater than
<	Is less than
==	Is equal to
<=	Is less than or equal to
>=	Is greater than or equal to
!=	Is not equal to

## Code

```
1 print(5 < 10)
2 print(2 > 1)
```

PYTHON

## Output

```
True
```

```
True
```

## Possible Mistakes

### Mistake - 1

#### Code

PYTHON

```
1 print(3 = 3)
```

#### Output

```
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

### Mistake - 2

#### Code

PYTHON

```
1 print(2 < = 3)
```

#### Output

```
SyntaxError: invalid syntax
```

Space between relational operators

`==` , `>=` , `<=` , `!=` is not valid in Python.

## Comparing Numbers

### Code

PYTHON

```
1 print(2 <= 3)
2 print(2.53 >= 2.55)
```

### Output

True

False

## Comparing Integers and Floats

### Code

PYTHON

```
1 print(12 == 12.0)
2 print(12 == 12.1)
```

### Output

True

False

## Comparing Strings

### Code

PYTHON

```
1 print("ABC" == "ABC")
2 print("CBA" != "ABC")
```

## Output

```
True
```

```
True
```

## Case Sensitive

### Code

```
PYTHON
```

```
1 print("ABC" == "abc")
```

## Output

```
False
```

Python is case sensitive.

It means

(Capital letter) and  (small letter) are not the same in Python.

## Strings and Equality Operator

### Code

```
PYTHON
```

```
1 print(True == "True")
2 print(123 == "123")
3 print(1.1 == "1.1")
```

## Output

False

False

False

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Logical Operators

The logical operators are used to perform logical operations on Boolean values.

Gives

True or False as the result.

Following are the logical operators

- and
- or
- not

### Logical AND Operator

Gives

True if both the booleans are true else, it gives False

#### Code

```
1 print(True and True)
```

PYTHON

#### Output

```
True
```

### Examples of Logical AND

Code

```
1 print((2 < 3) and (1 < 2))
```

PYTHON

*Step by Step Explanation*

(2 < 3) and (1 < 2)

True and (1 < 2)

True and True

Output

True

**Logical OR Operator**

Gives

True if any one of the booleans is true else, it gives False

Code

```
1 print(False or False)
```

PYTHON

Output

False

**Examples of Logical OR**

Code

```
1 print((2 < 3) or (2 < 1))
```

PYTHON

*Step by Step Explanation*

(2 < 3) or (2 < 1)

True or (2 < 1)

True or False

Output

True

## Logical NOT Operator

Gives the opposite value of the given boolean.

Code

```
1 print(not(False))
```

PYTHON

Output

True

## Examples of Logical NOT

Code

```
1 print(not(2 < 3))
```

#### Step by Step Explanation

```
not(2 < 3)
not(True)
False
```

#### Output

```
False
```

#### Summary

1. Logical *AND* Operator gives `True` if all the booleans are true.
2. Logical *OR* Operator gives `True` if any of the booleans are true.
3. Logical *NOT* Operator gives the opposite value



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Conditional Statements

#### Block of Code

A sequence of instructions are called block of code.

Python executes code in a sequence.

#### Condition

An expression that results in either

True    or    False

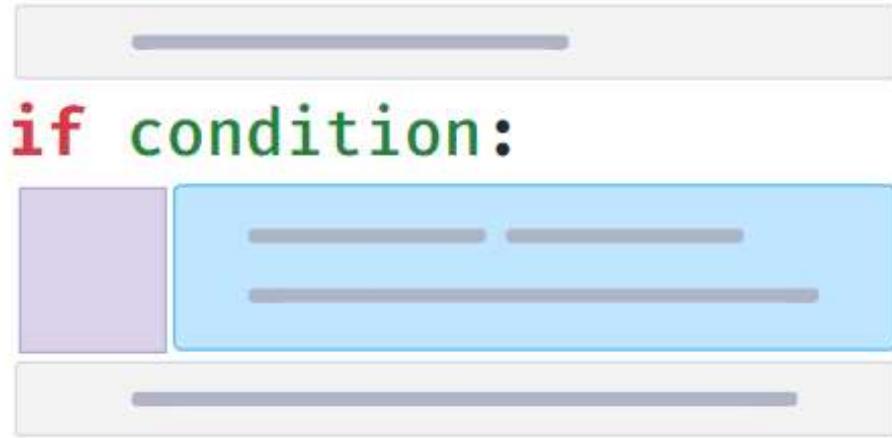
#### Examples

- i.  $2 < 3$
- ii.  $a == b$
- iii. True

### Conditional Statement

Conditional Statement allows you to execute a block of code only when a specific condition is

True

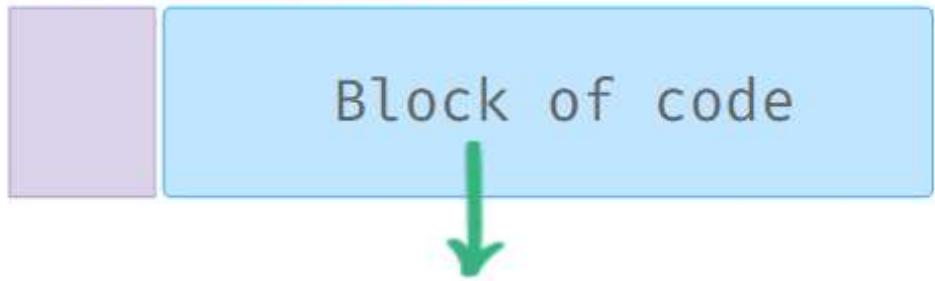


### `if condition:`

Block of code which executes only if a condition is

True is called **Conditional Block**.

### `if condition:`



### **Conditional Block**

Executes only if condition is true

Indentation

- Space(s) in front of the conditional block is called *indentation*.
- Indentation(spacing) is used to identify Conditional Block.
- Standard practice is to use *four spaces* for indentation.

Indentation  
Spacing is Used  
to Identify  
Conditional Block



#### Possible Mistakes

Each statement inside a conditional block should have the same indentation (spacing).

#### Wrong Code

```
1 if True:  
2     print("If Block")  
3         print("Inside If")
```

PYTHON

#### Output

```
IndentationError: unexpected indent
```

#### Correct Code

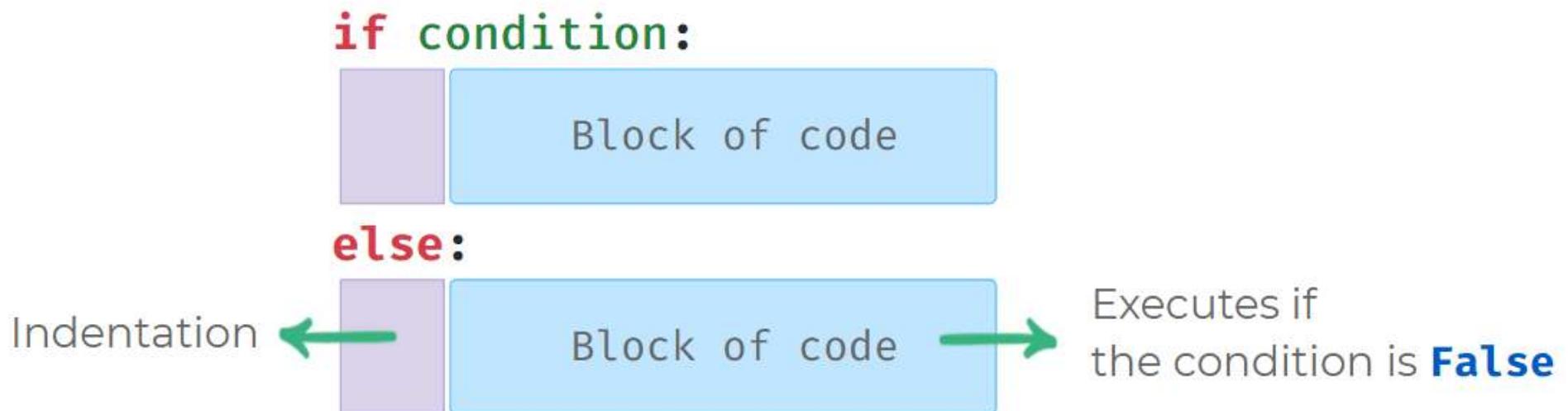
PYTHON

```
1 if True:  
2     print("If Block")  
3     print("Inside If")
```

## If - Else Syntax

When If - Else conditional statement is used, Else block of code executes if the condition is

False



## Using If-Else

### Code

PYTHON

```
1 a = int(input())  
2 if a > 0:  
3     print("Positive")  
4 else:  
5     print("Not Positive")  
6 print("End")
```

## Input

```
2
```

## Output

```
Positive
```

```
End
```

## Possible Mistakes in If-Else

Else can only be used along with if condition. It is written below if conditional block

## Code

PYTHON

```
1 ✘ if False:  
2     print("If Block")  
3     print("After If")  
4 ✘ else:  
5     print("Else Block")  
6     print("After Else")
```

## Output

```
SyntaxError: invalid syntax
```

 **Warning :**Note: No code is allowed in between if conditional block and else statement.

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### More Arithmetic Operators

#### Modulus

To find the remainder, we use Modulus operator

%

- a % b

#### Code

PYTHON

```
1 print(6 % 3)
```

#### Output

0

#### Exponent

To calculate a power b, we use Exponent Operator

\*\*

- a \*\* b

#### Code

PYTHON

```
1 print(2 ** 3)
```

## Output

```
8
```

You can use the exponent operator to calculate the square root of a number by keeping the exponent as

```
0.5
```

## Code

```
1 print(16 ** 0.5)
```

PYTHON

## Output

```
4.0
```



MARKED AS COMPLETE

[Submit Feedback](#)

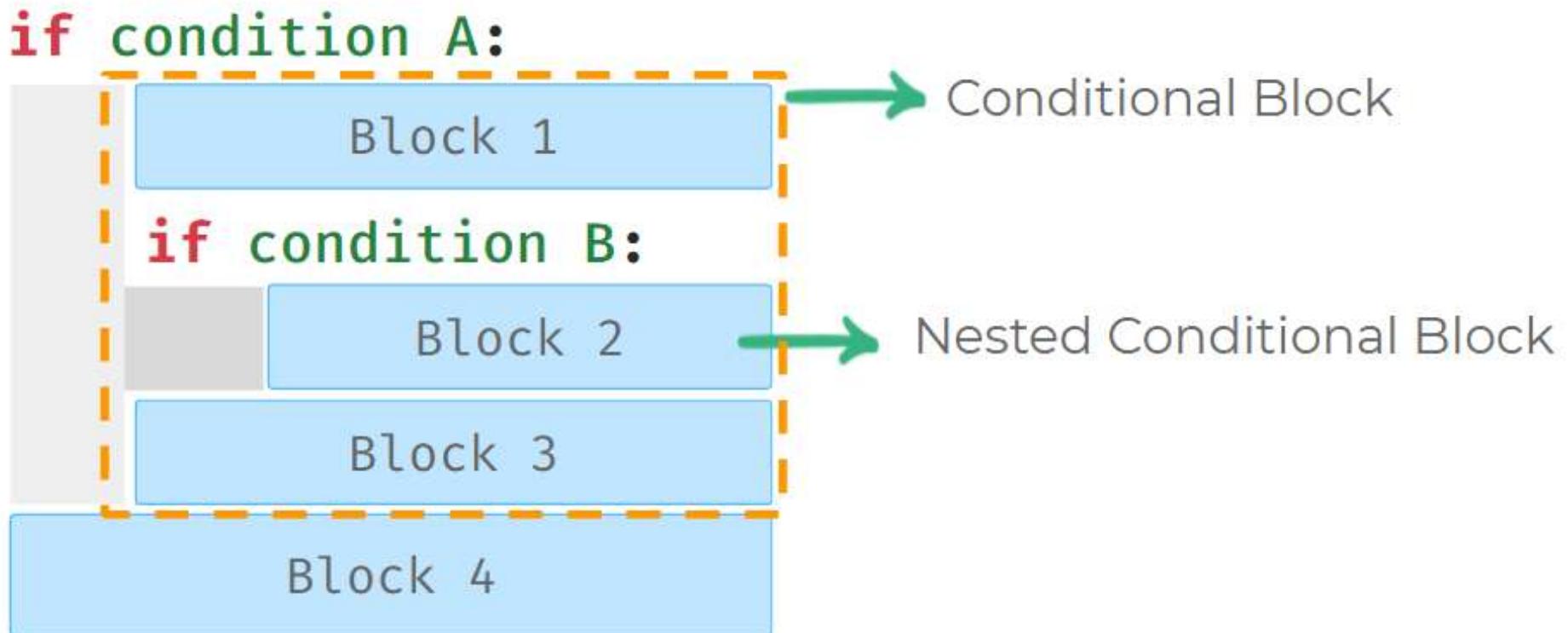
## Cheat Sheet

### Nested Conditional Statements

#### Nested Conditions

The conditional block inside another if/else conditional block is called as *nested conditional block*.

In the below example, **Block 2** is nested conditional block and **condition B** is called nested conditional statement.



#### Code

```
1 matches_won = int(input())
```

PYTHON

```
2 goals = int(input())
3 if matches_won > 8:
4     if goals > 20:
5         print("Hurray")
6     print("Winner")
```

### Input

```
10
22
```

### Output

```
Hurray
Winner
```

### Input

```
10
18
```

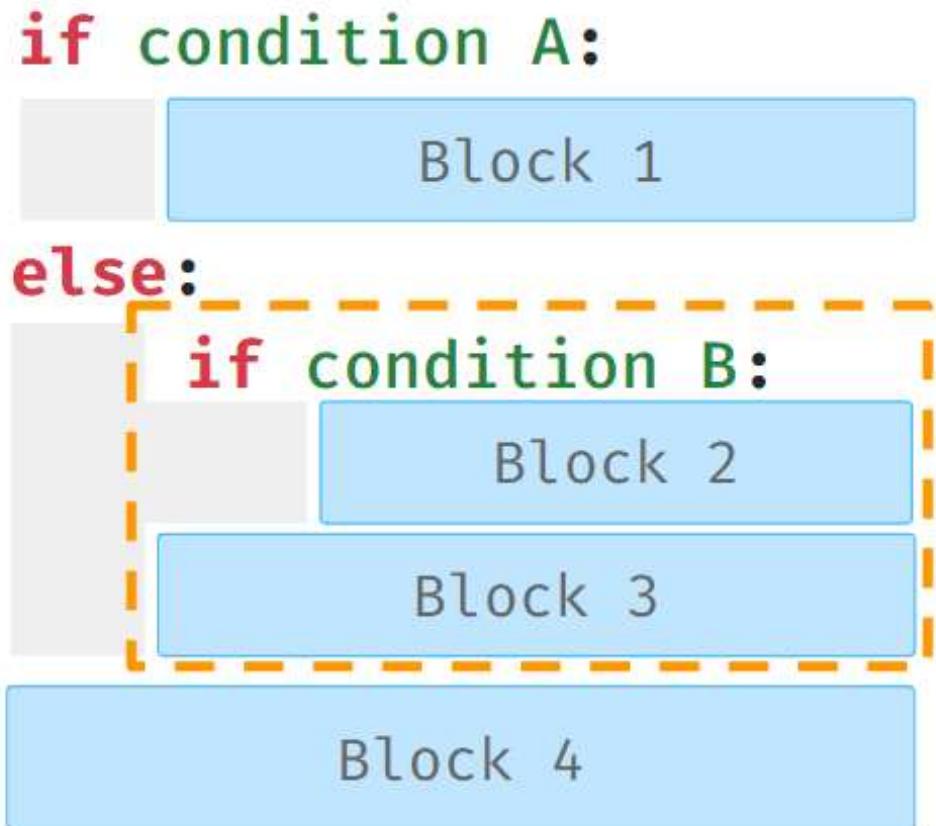
### Output

```
Winner
```

## Nested Condition in Else Block

We can also write nested conditions in Else Statement.

In the below example **Block 2** is a nested conditional block.



Code

```
1  a = 2  
2  b = 3  
3  c = 1  
4  is_a_greatest = (a > b) and (a > c)  
5  if is_a_greatest:  
6      print(a)  
7  else:  
8      is_b_greatest = (b > c)
```

PYTHON

```
9     if is_b_greatest:  
10        print(b)  
11    else:  
12        print(c)
```

Collapse ^

## Output

```
3
```

## Elif Statement

Use the elif statement to have multiple conditional statements between if and else.

The elif statement is optional.

```
if condition A:
```

```
    Block 1
```

```
elif condition B:
```

```
    Block 2
```

```
else:
```

```
    Block 3
```

#### Multiple Elif Statements

We can add any number of

**elif** statements after **if** conditional block.

```
if condition A:
```

```
    Block 1
```

```
elif condition B:
```

```
    Block 2
```

```
elif condition C:
```

```
    Block 3
```

```
else:
```

```
    Block 4
```

#### Execution of Elif Statement

Python will execute the elif block whose expression evaluates to true.

If multiple

elif conditions are true, then only the first elif block which is True will be executed.

**if** condition A:

Block 1

**elif** condition B:

Block 2

**elif** condition C:

Block 3

**else**:

Block 4

#### Optional Else Statement

Else statement is not compulsory after

if - elif statements.

**if** condition A:

Block 1

**elif** condition B:

Block 2

**elif** condition C:

Block 3

Code

PYTHON

```
1 number = 5
2 is_divisible_by_10 = (number % 10 == 0)
3 is_divisible_by_5 = (number % 5 == 0)
4 if is_divisible_by_10:
5     print("Divisible by 10")
6 elif is_divisible_by_5:
7     print("Divisible by 5")
8 else:
9     print("Not Divisible by 10 or 5")
```

Output

Divisible by 5

Possible Mistake

Cannot write an elif statement after

else statement.

**if condition A:**

Block 1

**elif condition B:**

Block 2

**else:**

Block 3

**elif condition C:**



Block 4

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Loops

So far we have seen that Python executes code in a sequence and each block of code is executed once.

**Loops allow us to execute a block of code several times.**

#### While Loop

Allows us to execute a block of code several times as long as the condition is

True .

## Initialization

→ **while** termination\_condition:



### While Loop Example

The following code snippet prints the next three consecutive numbers after a given number.

#### Code

PYTHON

```
1 a = int(input())
2 counter = 0
3 while counter < 3:
4     a = a + 1
5     print(a)
6     counter = counter + 1
```

Input

4

Output

5

6

7

Possible Mistakes

## 1. Missing Initialization

Code

PYTHON

```
1 a = int(input())
2 while counter < 3:
3     a = a + 1
4     print(a)
5     counter = counter + 1
6 print("End")
```

Input

5

Output

```
NameError: name 'counter' is not defined
```

## 2. Incorrect Termination Condition

Code

PYTHON

```
1 a = int(input())
2 counter = 0
3 condition = (counter < 3)
4 while condition:
5     a = a + 1
6     print(a)
7     counter = counter + 1
```

Input

```
10
```

Output

The above code runs into an infinite loop.

While block will keep repeating as the value in condition variable is

True .

## 3. Not Updating Counter Variable

Code

PYTHON

```
1 a = int(input())
2 counter = 0
3 while counter < 3:
4     a = a + 1
5     print(a)
6 print("End")
```

Input

10

Output

Infinite Loop

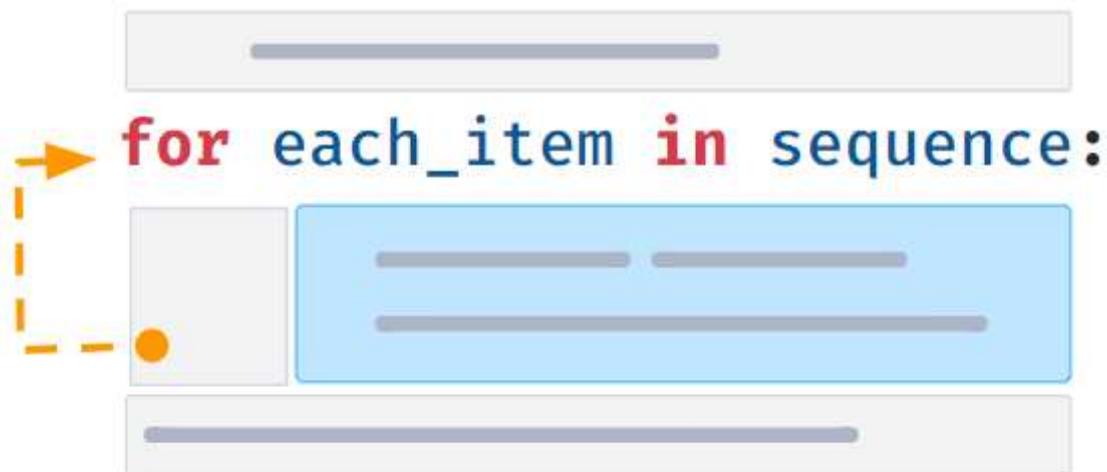
 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### For Loop

for statement iterates over each item of a sequence.



#### Examples of sequences:

- Sequence of Characters (string)
- Sequence of numbers, etc.

### For Syntax

#### Code

```
1 word = "Python"
2 for each_char in word:
3     print(each_char)
```

PYTHON

## Output

```
P  
y  
t  
h  
o  
n
```

## Range

Generates a sequence of integers starting from 0.

Syntax:

`range(n)` Stops before n (n is not included).

## Code

PYTHON

```
1 for number in range(3):  
2     print(number)
```

## Output

```
0  
1  
2
```

## Range with Start and End

Generates a sequence of numbers starting from

start

Syntax: `range(start, end)` Stops before `end` (end is not included).

Code

PYTHON

```
1 for number in range(5, 8):
2     print(number)
```

Output

```
5
6
7
```



MARKED AS COMPLETE

[Submit Feedback](#)

## Approach for Hollow pattern | Cheat Sheet

### Approach for Hollow pattern problem

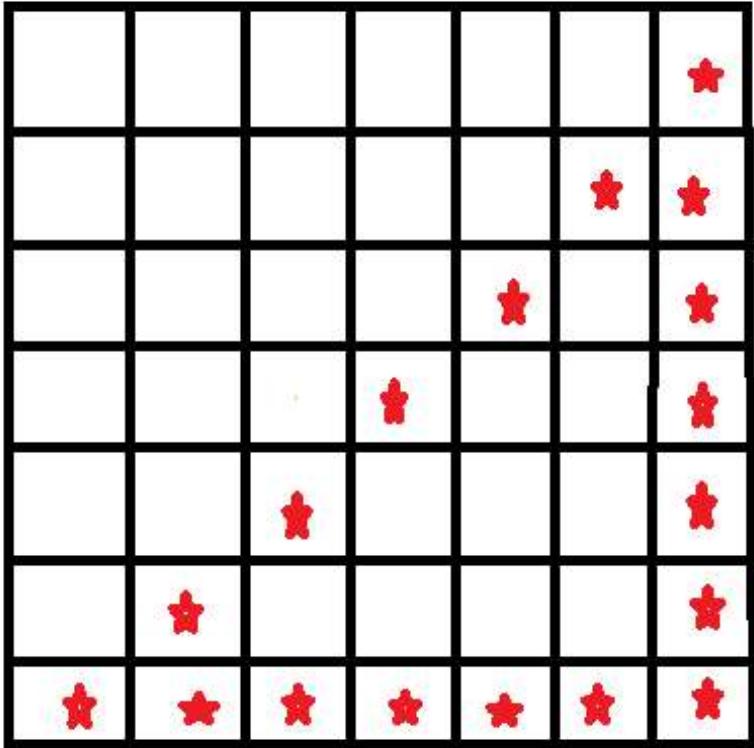
In this cheat sheet, let's see how to print the Hollow right-angled triangle pattern

Below is the pattern of the Hollow right-angled triangle

```
    *  
   * *  
  * * *  
 * * * *  
* * * * *  
* * * * * *  
* * * * * * *
```

Now, let's see how to print the 7x7 hollow right-angled triangle pattern.

Logical approach:



### Note

In the above image, each empty box contains 2 spaces (double space ' '), and each star is followed by a space ('\* ').

- While writing the code,
  - you need to print \* as '\* '(star followed by space).
  - you need to print 2 spaces(' ') in place of a single space.

The above image is the 7x7 pattern of the hollow right-angled triangle

- The first row has 6 spaces and has one star.

- The second row has 5 spaces and the first star and the second star.
- The third, fourth, fifth, and sixth rows have decreasing spaces, followed by the first star with increasing hollow spaces, then the second star.
- The last row has 7 stars.

Here in this problem

- space is denoted as the ' ' ( Double Spaces)
- star set denoted as the '\* ' (Star is followed by space)

Let's now discuss line by line

Pattern	Line Number	Number of Spaces			
*	First line	6 Double spaces	1st star		
**	Second line	5 Double spaces	1st star-set('* ')	2nd Star	
* *	Third line	4 Double spaces	1st star-set('* ')	1 Hollow spaces	2nd star
* *	Fourth line	3 Double spaces	1st star-set('* ')	2 Hollow spaces	2nd star
* *	Fifth line	2 Double spaces	1st star-set('* ')	3 Hollow spaces	2nd star
* * *	Sixth line	1 Double spaces	1st star-set('* ')	4 Hollow spaces	2nd star
* * * * * * *	Seventh line	0 Double spaces	7 star-sets('* ')		

Now, let's see how to print the above pattern using a python program.

- We can divide the above pattern logic into 3 parts
  - Part 1: we have **spaces** and **star('\*')** in first line
  - Part 2: we have **spaces, star, hollow spaces, star** (middle lines)
  - Part 3: we have **star set('\* ')** in the last line

## Code:

- We can use the For loop and if-elif-else statement to execute the particular part.

PYTHON

```
1 N = 7
2
3 for i in range(0, N):
4
5     if i == 0: # part- 1 logic
6
7         print(' '* (N-1) + '*')
8
9     elif i == N - 1: # part- 3 logic
10
11         print('* ' * N)
12
13 else: # part- 2 logic
14     left_spaces = ' '* (N -i-1)
15
16     hollow_spaces = (' ' * ( i - 1))
17
18     print(left_spaces + '* ' + hollow_spaces + '*')
```

Collapse ^

Now let's see the explanation of the for loop execution, step by step:

Part - 1 (first line):

i = 0

The first row has 6 spaces followed by one star. In order to get 6 no. of spaces, we must subtract 1 from the

N value.

```
spaces = ' ' * (N-1) = 6 // Here N value is 7
```

Output is

```
1 * * * *
```

Part - 2 (logic to print the Second line to Sixth line):

i = 1

In the second row, we have 5 spaces followed by the first star and hollow spaces followed by the second star. In order to get 5 no.of spaces, we must subtract 1 and i value from the N value.

```
spaces = ' ' * (N-i-1) = 5  
hollow_spaces = (' ' * ( i - 1)) = 0
```

Output is

```
* * * * *
```

i = 2

In the third row, we have 4 spaces followed by the first star and hollow spaces followed by the second star. In order to get 4 no.of spaces, we must subtract 1 and i value from the N value.

```
spaces = ' ' * (N-i-1) = 4  
hollow_spaces = (' ' * ( i - 1)) = 1
```

Output is

```
* * * * *
```

```
..... * *
```

i = 3

In the fourth row, we have 3 spaces followed by the first star and hallow spaces followed by the second star. In order to get 3 no.of spaces, we must subtract 1 and i value from the N value.

```
spaces = ' ' * (N-i-1) = 3  
hollow_spaces = (' ' * ( i - 1)) = 2
```

Output is

```
..... *  
..... * *  
..... * *  
..... * *
```

i = 4

In the fifth row, we have 2 spaces followed by the first star and hallow spaces followed by the second star. In order to get 2 no.of spaces, we must subtract 1 and i value from the N value.

```
spaces = ' ' * (N-i-1) = 2  
hollow_spaces = (' ' * ( i - 1)) = 3
```

Output is

```
..... *  
..... * *  
..... * *  
..... * *
```

```
..... * .. *
```

i = 5

In the sixth row, we have 1 space followed by the first star and hollow spaces followed by the second star. In order to get 1 space, we must subtract 1 and i value from the N value.

```
spaces = ' ' * (N-i-1) = 1  
hollow_spaces = (' ' * ( i - 1)) = 4
```

Output is

```
..... *  
..... * *  
..... * *  
..... * *  
..... * *  
..... * *
```

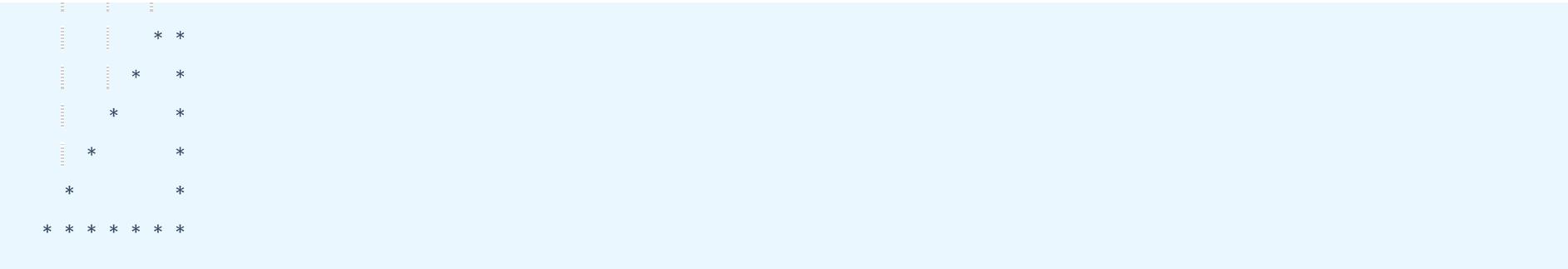
Part - 3 (Seventh line):

i = 6

```
stars = '*' * N
```

Output is

```
..... * .. *
```



MARKED AS COMPLETE

[Submit Feedback](#)

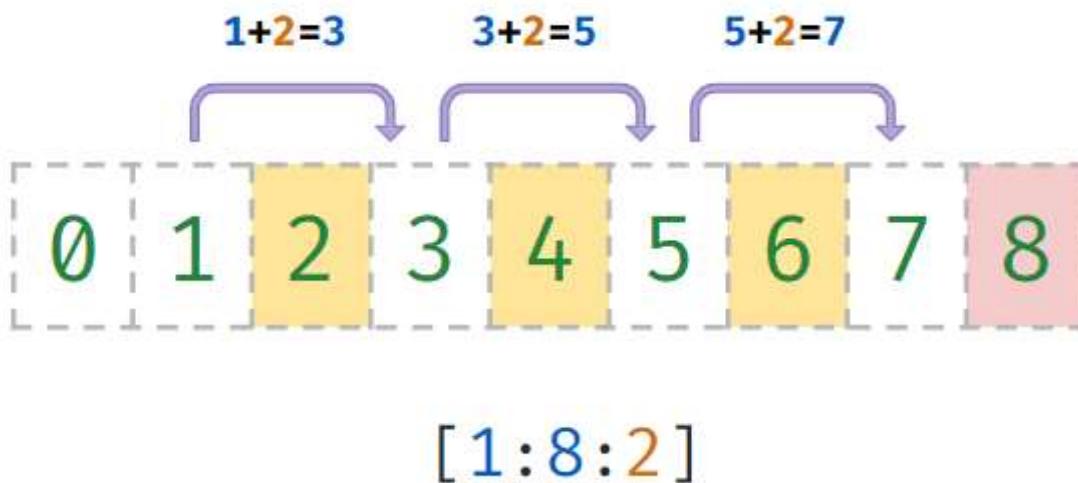
## Cheat Sheet

### Extended Slicing and String Methods

#### Extended Slicing

Syntax:

```
variable[start_index:end_index:step]
```



Step determines the increment between each index for slicing.

#### Code

```
1 a = "Waterfall"
2 part = a[1:6:3]
3 print(part)
```

PYTHON

## Output

```
ar
```

## Methods

Python has a set of built-in reusable utilities.

They simplify the most commonly performed operations are:

### String Methods

- `isdigit()`
- `strip()`
- `lower()`
- `upper()`
- `startswith()`
- `endswith()`
- `replace()` and more...

#### Isdigit

Syntax:

```
str_var.isdigit()
```

Gives

`True` if all the characters are digits. Otherwise, `False`

#### Code

```
1 is_digit = "4748".isdigit()  
2 print(is_digit)
```

PYTHON

## Output

```
True
```

## Strip

Syntax:

```
str_var.strip()
```

Removes all the leading and trailing spaces from a string.

## Code

PYTHON

```
1 mobile = " 9876543210 "
2 mobile = mobile.strip()
3 print(mobile)
```

## Output

```
9876543210
```

## Strip - Specific characters

Syntax:

```
str_var.strip(chars)
```

We can also specify characters that need to be removed.

## Code

```
1 name = "Ravi."
2 name = name.strip(".")
3 print(name)
```

## Output

```
Ravi
```

## Strip - Multiple Characters

Removes all spaces, comma(,) and full stop(.) that lead or trail the string.

## Code

```
1 name = ", , , ravi , , ."
2 name = name.strip(" ,.")
3 print(name)
```

## Output

```
ravi
```

## Replace

Syntax:

```
str_var.replace(old,new)
```

Gives a new string after replacing all the occurrences of the old substring with the new substring.

## Code

PYTHON

```
1 sentence = "teh cat and teh dog"
2 sentence = sentence.replace("teh","the")
3 print(sentence)
```

## Output

```
the cat and the dog
```

## Startswith

Syntax:

```
str_var.startswith(value)
```

Gives

True if the string starts with the specified value. Otherwise, False

## Code

PYTHON

```
1 url = "https://onthegomodel.com"
2 is_secure_url = url.startswith("https://")
3 print(is_secure_url)
```

## Output

```
True
```

## Endswith

Syntax:

```
str_var.endswith(value)
```

Gives

True if the string ends with the specified value. Otherwise, False

Code

PYTHON

```
1 gmail_id = "rahul123@gmail.com"
2 is_gmail = gmail_id.endswith("@gmail.com")
3 print(is_gmail)
```

Output

```
True
```

## Upper

Syntax:

```
str_var.upper()
```

Gives a new string by converting each character of the given string to uppercase.

Code

PYTHON

```
1 name = "ravi"
2 print(name.upper())
```

## Output

RAVI

## Lower

Syntax:

```
str_var.lower()
```

Gives a new string by converting each character of the given string to lowercase.

## Code

PYTHON

```
1 name = "RAVI"  
2 print(name.lower())
```

## Output

ravi



MARKED AS COMPLETE

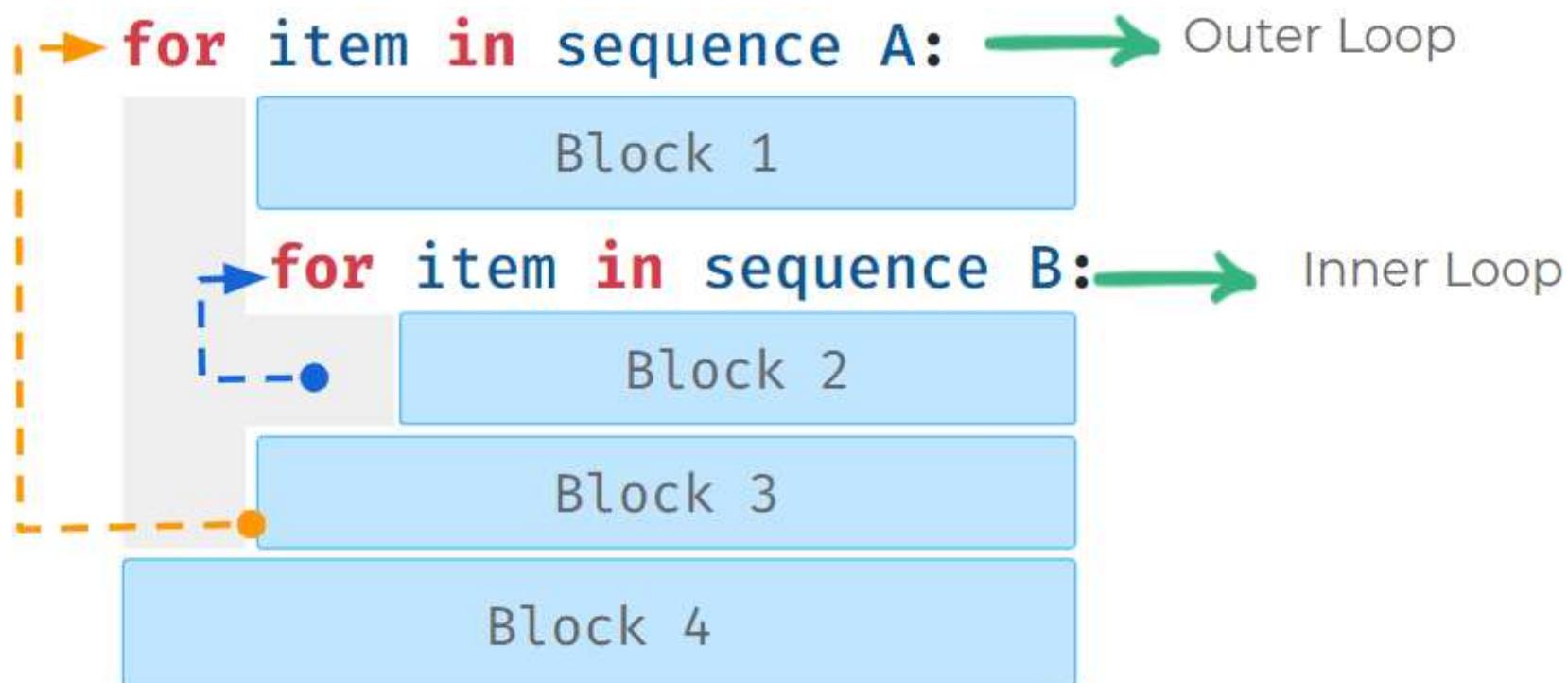
[Submit Feedback](#)

## Cheat Sheet

### Nested Loops

An inner loop within the repeating block of an outer loop is called Nested Loop.

The **Inner Loop** will be executed one time for each iteration of the **Outer Loop**.



## Code

```
1 for i in range(2):  
2     print("Outer: " + str(i))  
3     for j in range(2):
```

PYTHON

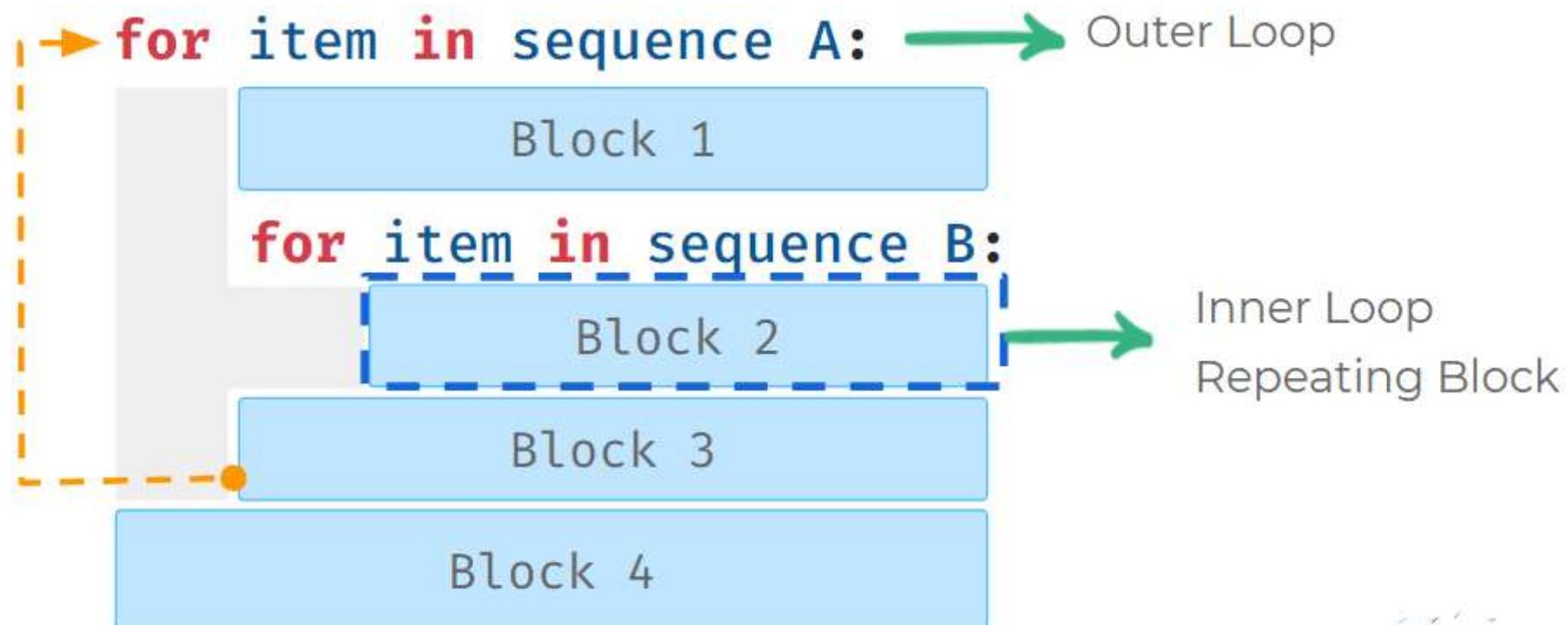
```
4     print(" Outer: " + str(i))
```

## Output

```
Outer: 0
Inner: 0
Inner: 1
Outer: 1
Inner: 0
Inner: 1
```

## Nested Repeating Block

The one highlighted in the blue dotted line is the **repeating** block of the inner loop.



## Code

PYTHON

```
1 for i in range(2):
2     print("Outer: " + str(i))
3 for j in range(2):
4     print(" Inner: " + str(j))
5 print("END")
```

In the above example, the below line is the repeating block of the nested loop.

## Code

PYTHON

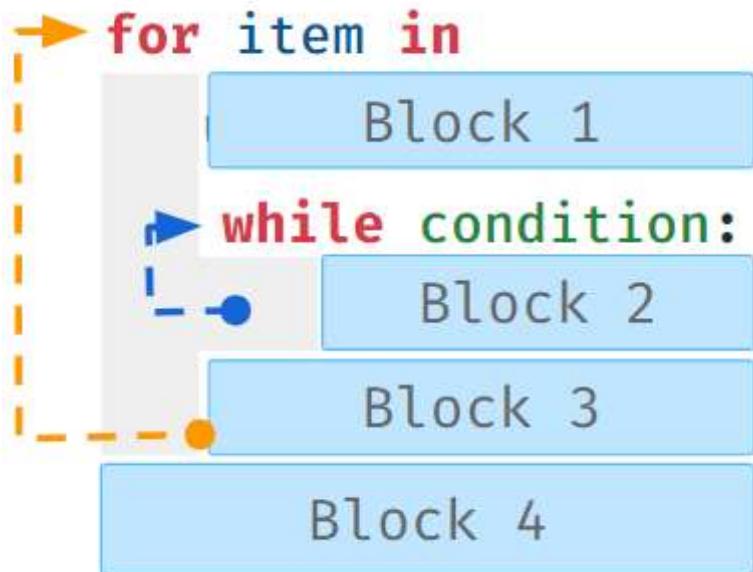
```
1 print(" Inner: " + str(j))
```

## Output

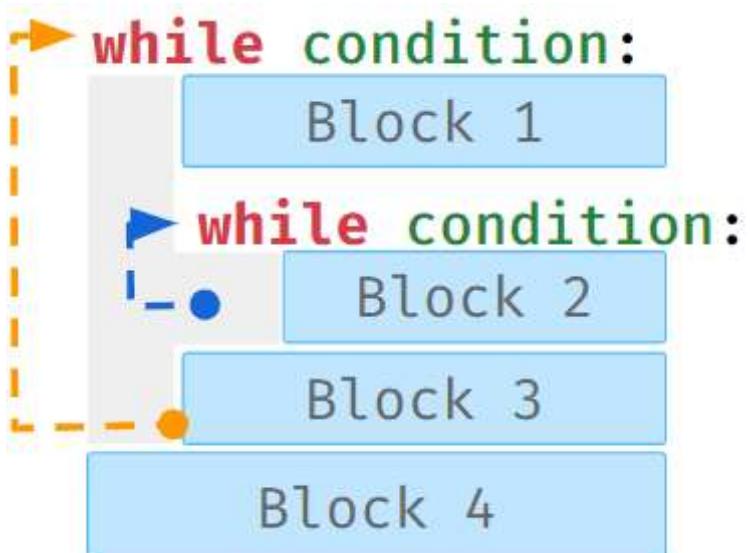
```
Outer: 0
Inner: 0
Inner: 1
Outer: 1
Inner: 0
Inner: 1
END
```

## Examples - Nested Loops

**Example - 1:** While loop inside a For loop



**Example - 2:** While loop inside a while loop





MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Loop Control Statements

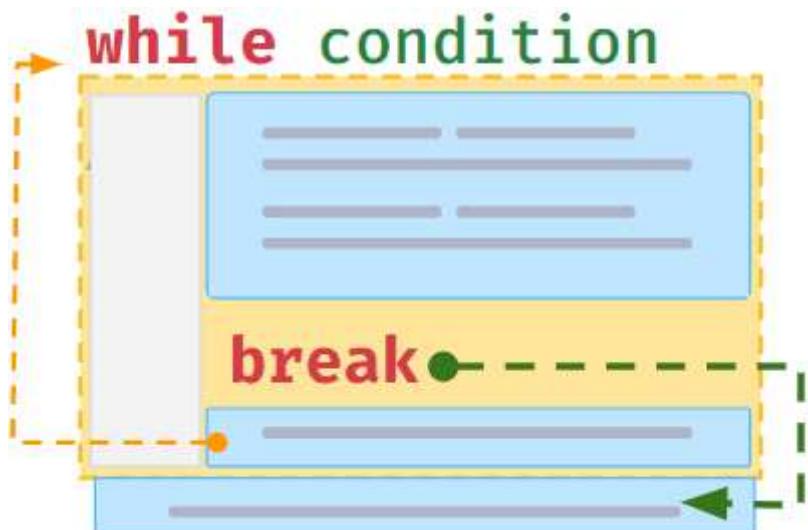
Control statements alter the sequential execution of a program.

#### Examples

- **if-elif-else**
- **while, for**
- **break, continue**

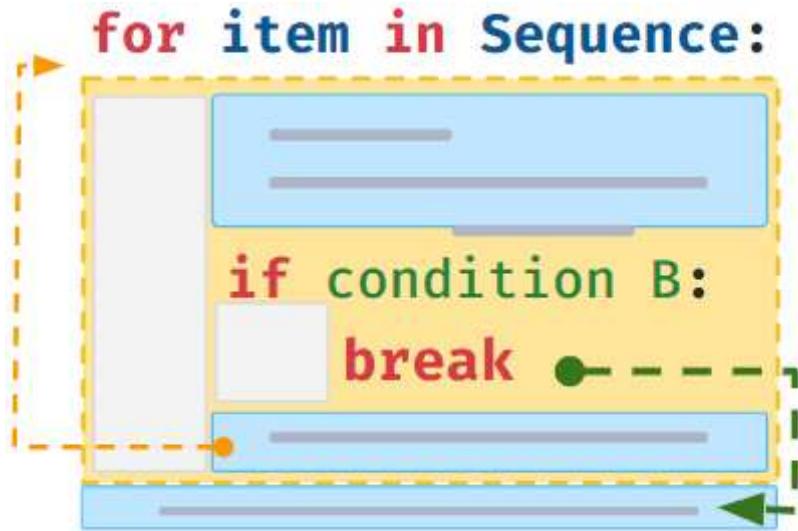
#### Break

Break statement makes the program exit a loop early.



#### Using Break

Generally, break is used to exit a loop when a condition is satisfied.



In the below example, when the variable

`i` value equals to `3` the `break` statement gets executed and stops the execution of the loop further.

Code

```
1 for i in range(5):
2     if i == 3:
3         break
4     print(i)
5 print("END")
```

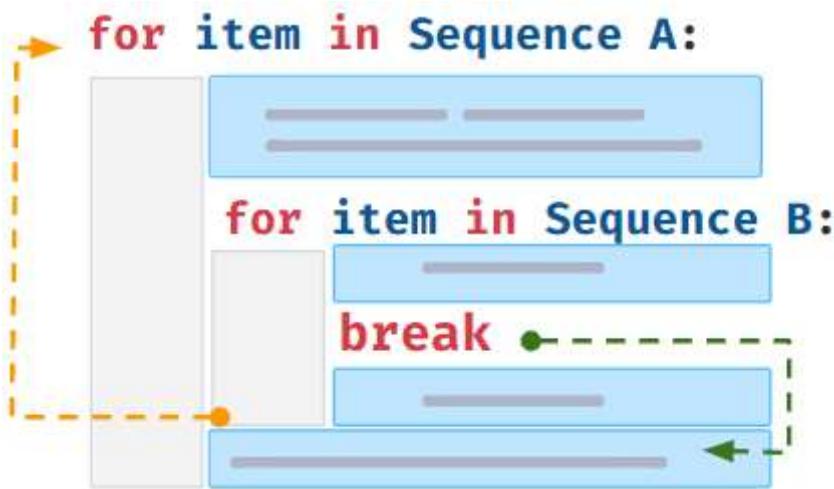
PYTHON

Output

```
0
1
2
END
```

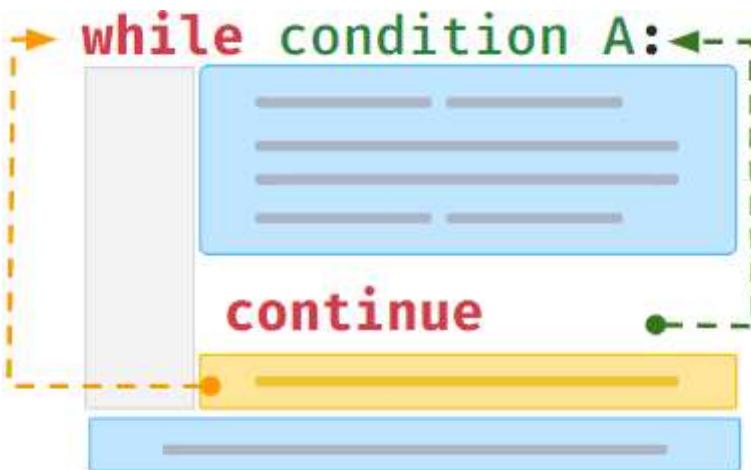
## Break in Nested Loop

Break in inner loop stops the execution of the inner loop.



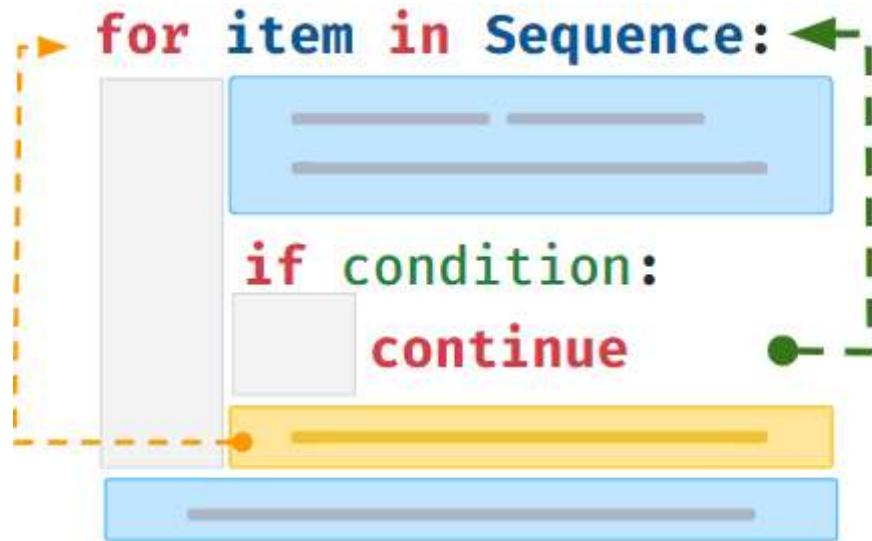
## Continue

Continue makes the program skip the remaining statements in the current iteration and begin the next iteration.



## Using Continue

Generally, continue is used to skip the remaining statements in the current iteration when a condition is satisfied.



In the below example, when the variable

i value equals to 3 the next statements in the loop body are skipped.

Code

PYTHON

```
1 for i in range(5):
2     if i == 3:
3         continue
4     print(i)
5 print("END")
```

Output

```
0
1
2
4
```

END

## Pass

Pass statement is used as a syntactic placeholder. When it is executed, nothing happens.

Generally used when we have to test the code before writing the complete code.

```
if condition A:  
    Block 1  
  
elif condition B:  
    pass  
  
else:  
    Block 3
```

## Empty Loops

We can use pass statements to test code written so far, before writing loop logic.

```
while condition A:    for item in Sequence:  
    pass                pass
```

 MARKED AS COMPLETE

Submit Feedback

## Cheat Sheet

### Comparing Strings

Computer internally stores characters as numbers.

Every character has a unique **Unicode** value.

"A"

"Z"

Unicode - 65

Unicode - 122

"1"

"\*"

Unicode - 49

Unicode - 42

### Ord

To find the Unicode value of a character, we use the

ord()

ord(character) gives unicode value of the character.

### Code

```
1 unicode_value = ord("A")
2 print(unicode_value)
```

PYTHON

## Output

```
65
```

## chr

To find the character with the given Unicode value, we use the

```
chr()
```

```
chr(unicode)
```

 gives character with the unicode value.

## Code

PYTHON

```
1 char = chr(75)
2 print(char)
```

## Output

```
K
```

## Unicode Ranges

**48 - 57** -> Number Digits (0 - 9)

**65 - 90** -> Capital Letters (A - Z)

**97 - 122** -> Small Letters (a - z)

## Printing Characters

The below code will print the characters from

A to Z

### Code

PYTHON

```
1 for unicode_value in range(65,91):
2     print(chr(unicode_value))
```

### Output

A  
B  
C  
D  
E  
F  
G  
H  
I  
J  
K  
L  
M  
N  
O  
P  
Q  
R

```
''  
S  
T  
U  
V  
W  
X  
Y  
Z
```

Collapse ^

## Comparing Strings

In Python, strings are compared considering unicode.

### Code

```
1 print("A" < "B")
```

PYTHON

### Output

```
True
```

As unicode value of

A is 65 and B is 66, which internally compares 65 < 66 . So the output should be True

## Character by Character Comparison

In Python, String Comparison is done character by character.

### Code

PYTHON

```
1 print("BAD" >= "BAT")
```

Output

```
False
```

Code

```
1 print("98" < "984")
```

PYTHON

Output

```
True
```

## Best Practices

### Naming Variables Rule #1

Use only the below characters

- Capital Letters ( A – Z )
- Small Letters ( a – z )
- Digits ( 0 – 9 )
- Underscore( )

*Examples:*

age, total\_bill

### Naming Variables Rule #2

Below characters cannot be used

- Blanks ( )
- Commas ( , )
- Special Characters  
( ~ ! @ # \$ % ^ . ?, etc. )

### Naming Variables Rule #3

Variable name must begin with

- Capital Letters ( A – Z )
- Small Letters ( a – z )
- Underscore( \_ )

### Naming Variables Rule #4

Cannot use Keywords, which are reserved for special meaning

- int
- str
- print etc.,

## Keywords

Words which are reserved for special meaning

### Code

PYTHON

```
1 help("keywords")
```

### Output

Here is a list of the Python keywords. Enter any keyword to get more help.

False	break	for	not
None	class	from	or

True	continue	global	pass
<u>__peg_parser__</u>	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield

None

Collapse ^

## Case Styles

- Camel case: **totalBill**
- Pascal case: **TotalBill**
- Snake case: **total\_bill**

Snake case is preferred for naming the variables in Python.



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Round

#### Rounding Numbers

`round(number, digits(optional))` Rounds the float value to the given number of decimal digits.

`digits` -> define the number of decimal digits to be considered for rounding.

- when not specified default is

0

#### Code

```
1 a = round(3.14,1)
2 print(a)
3 a = round(3.14)
4 print(a)
```

PYTHON

#### Output

```
3.1
3
```

### Floating Point Approximation

Float values are stored approximately.

#### Code

```
print(0.1 + 0.2)
```

Output

```
0.3000000000000004
```

## Floating Point Errors

Sometimes, floating point approximation gives unexpected results.

Code

PYTHON

```
1 print((0.1 + 0.2) == 0.3)
```

Output

```
False
```

To avoid these unexpected results, we can use

round()

Code

PYTHON

```
1 a = round((0.1 + 0.2), 1)
2 print(a)
3 print(a == 0.3)
```

## Output

0.3

True

## Comments

Comment starts with a hash

#

It can be written in its own line next to a statement of code.

## Code

PYTHON

```
1 n = 5
2 # Finding if Even
3 even = (n % 2 == 0)
4 print(even) # prints boolean value
```

## Output

False

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Floor Division Operator

To find integral part of quotient we use Floor Division Operator

//

- a // b

Code

PYTHON

```
1 print(3 // 2)
```

Output

```
1
```

### Compound Assignment Operators

a  $+=$  1



Compound

## Assignment Operator

Different compound assignment operators are

$+=$  ,  $-=$  ,  $*=$  ,  $/=$  ,  $\%=$

$a += 1$  is similar to  $a = a + 1$

Code

PYTHON

```
1 a = 10
2 a -= 2
3 print(a)
```

Output

8

### Examples of Compound Assignment Operators

Code

PYTHON

```
1 a = 10
2 a /= 2
3 print(a)
```

Output

```
5.0
```

Code

```
1 a = 10
2 a %= 2
3 print(a)
```

PYTHON

Output

```
0
```

## Escape Characters

### Single And Double Quotes

String is a sequence of characters enclosed within quotes.

Code

```
1 sport = 'Cricket'
2 print(type(sport))
3 sport = "Cricket"
4 print(type(sport))
```

PYTHON

## Output

```
<class 'str'>  
<class 'str'>
```

## Code

PYTHON

```
1 is_same = ('Cricket' == "Cricket")  
2 print(is_same)
```

## Output

```
True
```

## Escape Characters

Escape Characters are a sequence of characters in a string that are interpreted differently by the computer.

We use escape characters to insert characters that are illegal in a string.

## Code

PYTHON

```
1 print("Hello\nWorld")
```

## Output

```
Hello  
World
```

We got a new line by adding

\n escape character.

### Examples - Escape Characters

Escape Characters start with a backslash in Python

- \n -> New Line
- \t -> Tab Space
- \\ -> Backslash
- \' -> Single Quote
- \" -> Double Quote

### Passing Strings With Quotes

The backslash

\ character here tells Python not to consider the next character as the ending of the string.

### Code

```
1 print('It\'s Python')
```

PYTHON

### Output

```
It's Python
```

### Code

```
1 print("It's Python")
```

PYTHON

## Output

```
It's Python
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Data Structures

Data Structures allow us to store and organize data efficiently.

This will allow us to easily access and perform operations on the data.

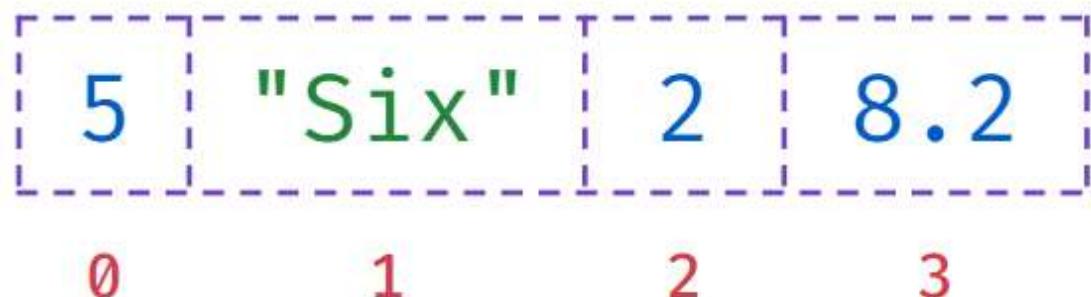
In Python, there are four built-in data structures

- *List*
- *Tuple*
- *Set*
- *Dictionary*

#### List

List is the most versatile python data structure.

Holds an ordered sequence of items.



#### Creating a List

Created by enclosing elements within [square] brackets.

Each item is separated by a comma.

## Code

PYTHON

```
1 a = 2
2 list_a = [5, "Six", a, 8.2]
3 print(type(list_a))
4 print(list_a)
```

## Output

```
<class 'list'>
[5, 'Six', 2, 8.2]
```

## Creating a List of Lists

### Code

PYTHON

```
1 a = 2
2 list_a = [5, "Six", a, 8.2]
3 list_b = [1, list_a]
4 print(list_b)
```

## Output

```
[1, [5, 'Six', 2, 8.2]]
```

## Length of a List

### Code

PYTHON

```
1 a = 2
2 list_a = [5, "Six", a, 8.2]
```

```
3 print(len(list_a))
```

Output

```
4
```

### Accessing List Items

To access elements of a list, we use **Indexing**.

Code

PYTHON

```
1 a = 2
2 list_a = [5, "Six", a, 8.2]
3 print(list_a[1])
```

Output

```
Six
```

### Iterating Over a List

Code

PYTHON

```
1 a = 2
2 list_a = [5, "Six", a, 8.2]
3 for item in list_a:
4     print(item)
```

Output

```
5
```

```
Six
```

```
2
```

```
8.2
```

## List Concatenation

Similar to strings,

+ operator concatenates lists.

### Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_b = ["a", "b", "c"]
3 list_c = list_a + list_b
4 print(list_c)
```

### Output

```
[1, 2, 3, 'a', 'b', 'c']
```

## Adding Items to List

### Code

PYTHON

```
1 list_a = []
2 print(list_a)
3 for i in range(1,4):
4     list_a += [i]
5 print(list_a)
```

## Output

```
[]  
[1, 2, 3]
```

## Repetition

- \* Operator repeats lists.

## Code

PYTHON

```
1 list_a = [1, 2]  
2 list_b = list_a * 3  
3 print(list_b)
```

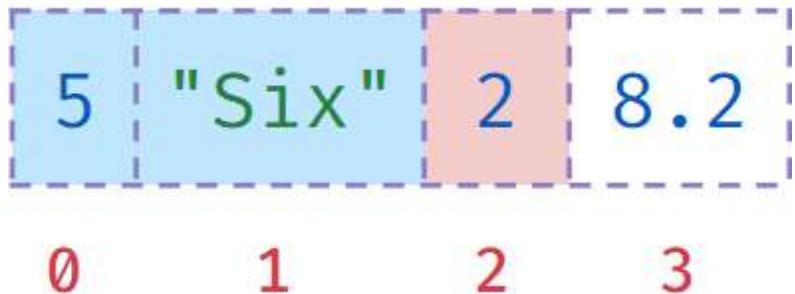
## Output

```
[1, 2, 1, 2, 1, 2]
```

## List Slicing

Obtaining a part of a list is called List Slicing.

# list\_a



Code

PYTHON

```
1 list_a = [5, "Six", 2, 8.2]
2 list_b = list_a[:2]
3 print(list_b)
```

Output

```
[5, 'Six']
```

Extended Slicing

Similar to string extended slicing, we can extract alternate items using step.

Code

PYTHON

```
1 list_a = ["R", "B", "G", "O", "W"]
2 list_b = list_a[0:5:3]
3 print(list_b)
```

Output

```
[ 'R', 'O' ]
```



## Converting to List

`list(sequence)` takes a sequence and converts it into list.

### Code

PYTHON

```
1 color = "Red"
2 list_a = list(color)
3 print(list_a)
```

### Output

```
[ 'R', 'e', 'd' ]
```

### Code

PYTHON

```
1 list_a = list(range(4))
2 print(list_a)
```

### Output

```
[0, 1, 2, 3]
```

## Lists are Mutable

- Lists can be modified.
- Items at any position can be updated.

#### Code

PYTHON

```
1 list_a = [1, 2, 3, 5]
2 print(list_a)
3 list_a[3] = 4
4 print(list_a)
```

#### Output

```
[1, 2, 3, 5]
[1, 2, 3, 4]
```

#### Strings are Immutable

Strings are Immutable (Can't be modified).

#### Code

PYTHON

```
1 message = "sea you soon"
2 message[2] = "e"
3 print(message)
```

#### Output

```
TypeError: 'str' object does not support item assignment
```



MARKED AS COMPLETE

Submit Feedback

## Cheat Sheet

### Working with Lists

#### Object

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

*Strings, Integers, Floats, Lists etc.* are all objects.

#### Examples

- "A"
- 1.25
- [1,2,3]

#### 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.

"A"

Id - 140035229724336



Id - 139630925071104

Finding Id

We can use the

`id()` to find the id of a object.

Code

```
1 print(id("Hello"))
```

PYTHON

Output

```
140589419285168
```

Id of Lists

PYTHON

```
1 list_a = [1, 2, 3]
2 list_b = [1, 2, 3]
3 print(id(list_a))
4 print(id(list_b))
```

## Output

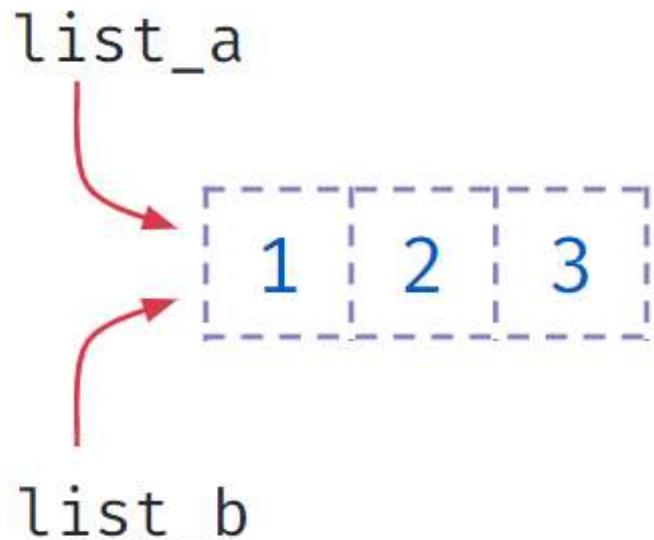
```
139637858236800
139637857505984
```

## Modifying Lists

### Modifying Lists - 1

When assigned an existing list both the variables

list\_a and list\_b will be referring to the same object.



## Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_b = list_a
3 print(id(list_a))
4 print(id(list_b))
```

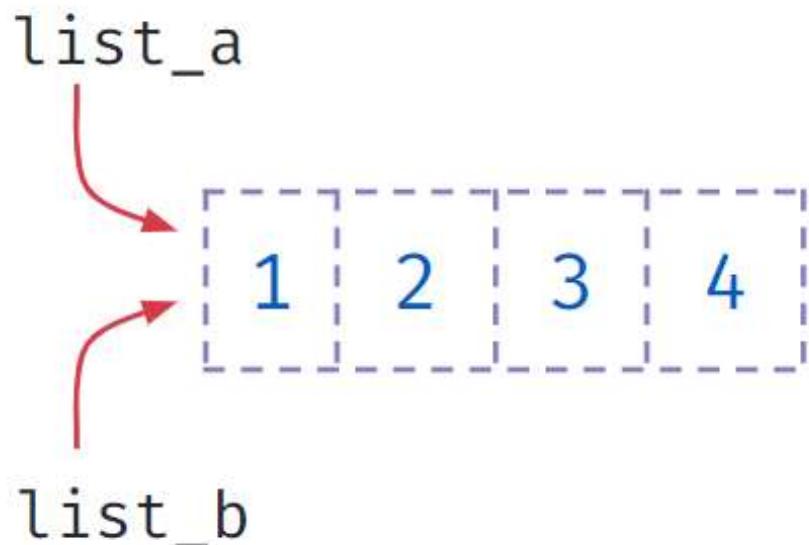
## Output

```
140334087715264
140334087715264
```

## Modifying Lists - 2

When assigned an existing list both the variables

list\_a and list\_b will be referring to the same object.



## Code

PYTHON

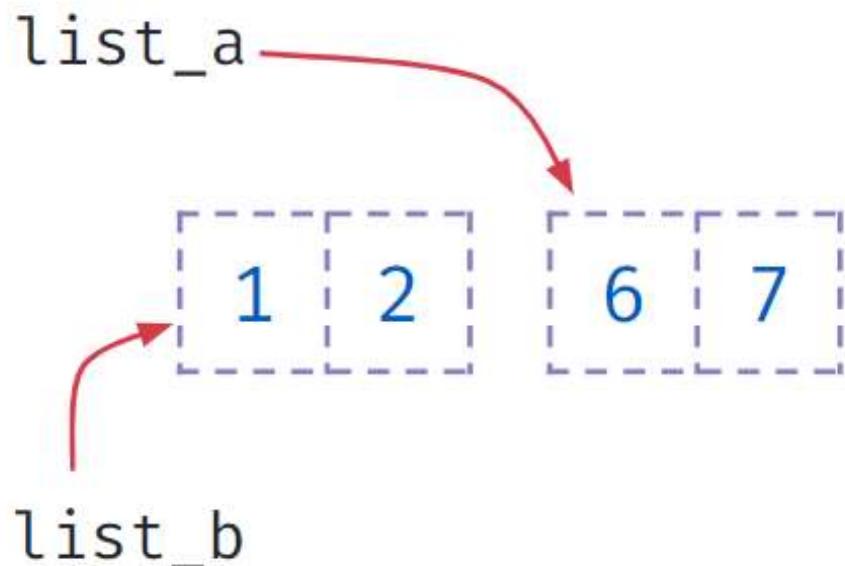
```
1 list_a = [1, 2, 3, 5]
2 list_b = list_a
3 list_b[3] = 4
4 print("list a : " + str(list_a))
5 print("list b : " + str(list_b))
```

## Output

```
list a : [1, 2, 3, 4]
list b : [1, 2, 3, 4]
```

## Modifying Lists - 3

The assignment will update the reference to new object.



## Code

PYTHON

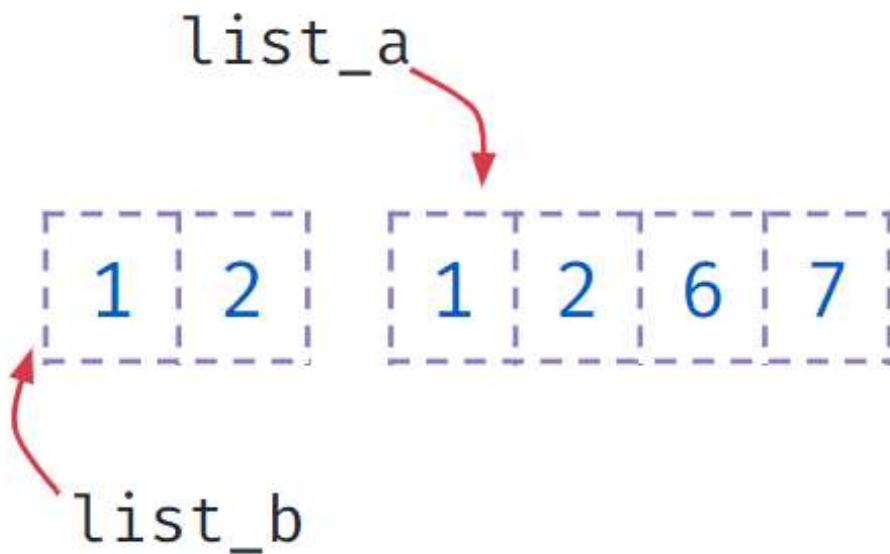
```
1 list_a = [1, 2]
2 list_b = list_a
3 list_a = [6, 7]
4 print("list a : " + str(list_a))
5 print("list b : " + str(list_b))
```

## Output

```
list a : [6, 7]
list b : [1, 2]
```

## Modifying Lists - 4

The assignment will update the reference to a new object.



## Code

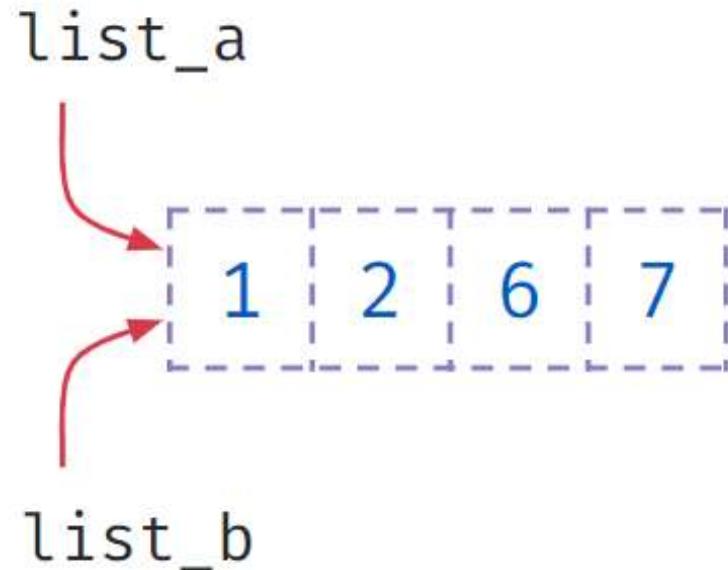
```
1 list_a = [1, 2]
2 list_b = list_a
3 list_a = list_a + [6, 7]
4 print("list a : " + str(list_a))
5 print("list b : " + str(list_b))
```

## Output

```
list a : [1, 2, 6, 7]
list b : [1, 2]
```

## Modifying Lists - 5

Compound assignment will update the existing list instead of creating a new object.



## Code

```
1 list_a = [1, 2]
2 list_b = list_a
3 list_a += [6, 7]
4 print("list a : " + str(list_a))
5 print("list b : " + str(list_b))
```

## Output

```
list a : [1, 2, 6, 7]
list b : [1, 2, 6, 7]
```

## Modifying Lists - 6

Updating mutable objects will also effect the values in the list, as the reference is changed.

list\_a



list\_b

Code

PYTHON

```
1 list_a = [1,2]
2 list_b = [3, list_a]
3 list_a[1] = 4
4 print(list_a)
5 print(list_b)
```

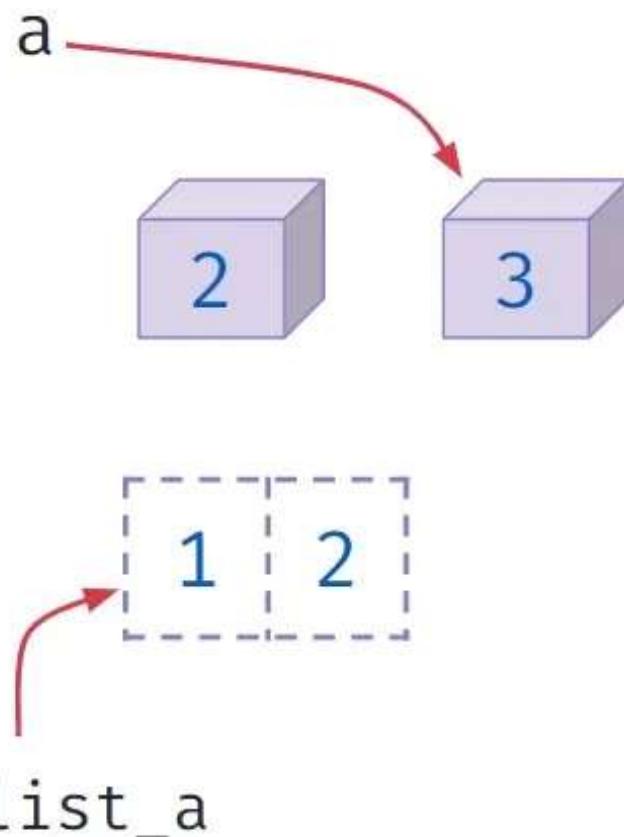
Output

```
[1, 4]
```

```
[3, [1, 4]]
```

## Modifying Lists - 7

Updating immutable objects will not effect the values in the list, as the reference will be changed.



### Code

PYTHON

```
1 a = 2
2 list_a = [1,a]
3 print(list_a)
4 a = 3
5 print(list_a)
```

## Output

```
[1, 2]
```

```
[1, 2]
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Lists and Strings

#### Splitting

```
str_var.split(separator)
```

Splits a string into a list at every specified separator.

If no separator is specified, default separator is whitespace.

#### Code

PYTHON

```
1  nums = "1 2 3 4"
2  num_list = nums.split()
3  print(num_list)
```

#### Output

```
['1', '2', '3', '4']
```

#### Multiple WhiteSpaces

Multiple whitespaces are considered as single when splitting.

#### Code

PYTHON

```
1  nums = "1 2 3 4 "
2  num_list = nums.split()
3  print(num_list)
```

## Output

```
['1', '2', '3', '4']
```

## New line

\n and tab space \t are also whitespace.

## Code

PYTHON

```
1  nums = "1\n2\t3 4"
2  num_list = nums.split()
3  print(num_list)
```

## Output

```
['1', '2', '3', '4']
```

## Using Separator

Breaks up a string at the specified separator.

*Example -1*

## Code

PYTHON

```
1  nums = "1,2,3,4"
2  num_list = nums.split(',')
3  print(num_list)
```

## Output

```
['1', '2', '3', '4']
```

### Example -2

#### Code

PYTHON

```
1 nums = "1,2,,3,4,"  
2 num_list = nums.split(',')  
3 print(num_list)
```

#### Output

```
['1', '2', '', '3', '4', '']
```

### Space as Separator

#### Code

PYTHON

```
1 nums = "1 2 3 4 "  
2 num_list = nums.split(" ")  
3 print(num_list)
```

#### Output

```
['1', '', '2', '3', '4', '']
```

### String as Separator

*Example - 1*

Code

PYTHON

```
1 string_a = "Python is a programming language"
2 list_a = string_a.split('a')
3 print(list_a)
```

Output

```
['Python is ', ' progr', 'mming l', 'ngu', 'ge']
```

*Example - 2*

PYTHON

```
1 string_a = "step-by-step execution of code"
2 list_a = string_a.split('step')
3 print(list_a)
```

Output

```
 ['', '-by-', ' execution of code']
```

Joining

```
str.join(sequence)
```

Takes all the items in a sequence of strings and joins them into one string.

Code

PYTHON

```
1 list_a = ['Python is ', ' programming ', 'a', 'language']
2 string_a = "a".join(list_a)
3 print(string_a)
```

## Output

```
Python is a programming language
```

## Joining Non String Values

Sequence should not contain any non-string values.

## Code

```
1 list_a = list(range(4))
2 string_a = ",".join(list_a)
3 print(string_a)
```

PYTHON

## Output

```
TypeError: sequence item 0: expected str instance, int found
```

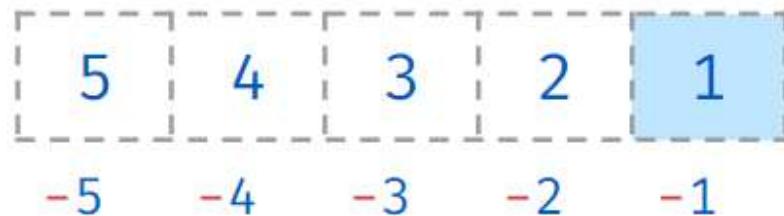
## Negative Indexing

Using a negative index returns the nth item from the end of list.

Last item in the list can be accessed with index

-1

list\_a[-1]



Reversing a List

-1 for step will reverse the order of items in the list.

Code

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
2 list_b = list_a[::-1]
3 print(list_b)
```

Output

```
[1, 2, 3, 4, 5]
```

Accessing List Items

### *Example-1*

Code

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
2 item = list_a[-1]
3 print(item)
```

Output

```
1
```

### *Example-2*

Code

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
2 item = list_a[-4]
3 print(item)
```

Output

```
4
```

### Slicing With Negative Index

You can also specify negative indices while slicing a List.

Code

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
```

```
2 list_b = list_a[-3:-1]
3 print(list_b)
```

Output

```
[3, 2]
```

list\_a[-3:-1]



-5    -4    -3    -2    -1



Out of Bounds Index

While slicing, Index can go out of bounds.

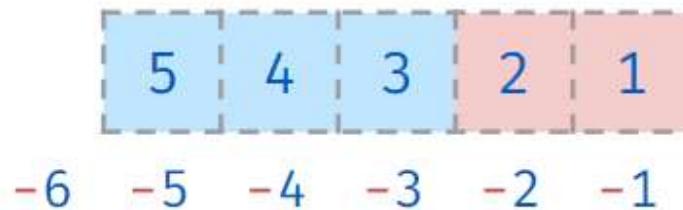
Code

```
1 list_a = [5, 4, 3, 2, 1]
2 list_b = list_a[-6:-2]
3 print(list_b)
```

PYTHON

## Output

```
[5, 4, 3]
```



## Negative Step Size

```
variable[start:end:negative_step]
```

Negative Step determines the decrement between each index for slicing.

Start index should be greater than the end index in this case

- start > end

## Negative Step Size Examples

### *Example - 1*

#### Code

```
1 list_a = [5, 4, 3, 2, 1]
2 list_b = list_a[4:2:-1]
3 print(list_b)
```

PYTHON

## Output

```
[1, 2]
```

### Example - 2

Negative step requires the start to be greater than end.

#### Code

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
2 list_b = list_a[2:4:-1]
3 print(list_b)
```

#### Output

```
[]
```

### Reversing a List

-1 for step will reverse the order of items in the list.

PYTHON

```
1 list_a = [5, 4, 3, 2, 1]
2 list_b = list_a[::-1]
3 print(list_b)
```

#### Output

```
[1, 2, 3, 4, 5]
```

## Reversing a String

-1 for step will reverse the order of the characters.

### Code

PYTHON

```
1 string_1 = "Program"  
2 string_2 = string_1[::-1]  
3 print(string_2)
```

### Output

```
margorP
```

## Negative Step Size - Strings

### Code

PYTHON

```
1 string_1 = "Program"  
2 string_2 = string_1[6:0:-2]  
3 print(string_2)
```

### Output

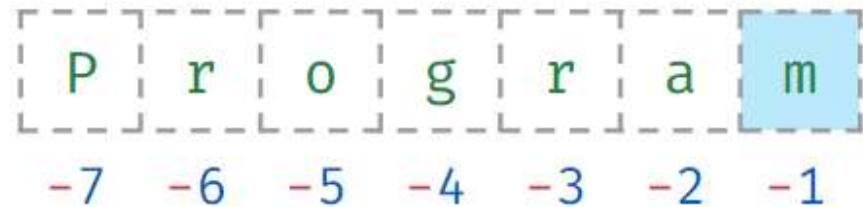
```
mro
```

## Indexing & Slicing - Strings

### Example - 1

### Code

## string\_1[-1]



PYTHON

```
1 string_1 = "Program"
2 string_2 = string_1[-1]
3 print(string_2)
```

Output

m

Example - 2

Code

```
1 string_1 = "Program"
2 string_2 = string_1[-4:-1]
3 print(string_2)
```

PYTHON

Output

gra

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

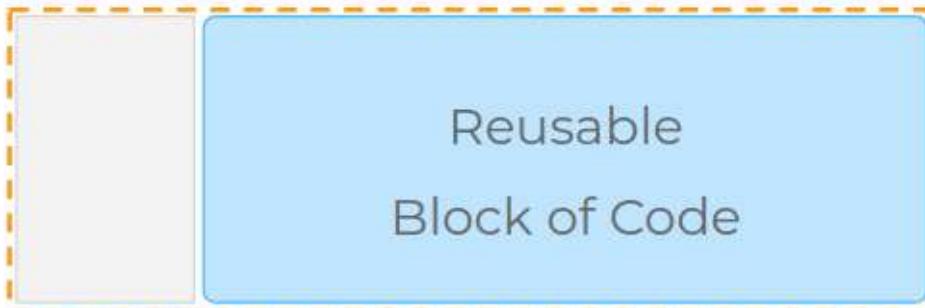
### Functions

Block of reusable code to perform a specific action.

#### Reusing Code

Using an existing code without writing it every time we need.

**def function\_name():**



#### Code

PYTHON

```
1 def greet():
2     print("Hello")
3
4 name = input()
5 print(name)
```

#### Input

Teja

Output

Teja

Defining a Function

Function is uniquely identified by the

function\_name

**def function\_name():**

Reusable  
Block of Code

Code

PYTHON

```
1 def greet():
2     print("Hello")
3
4 name = input()
5 print(name)
```

Input

Teja

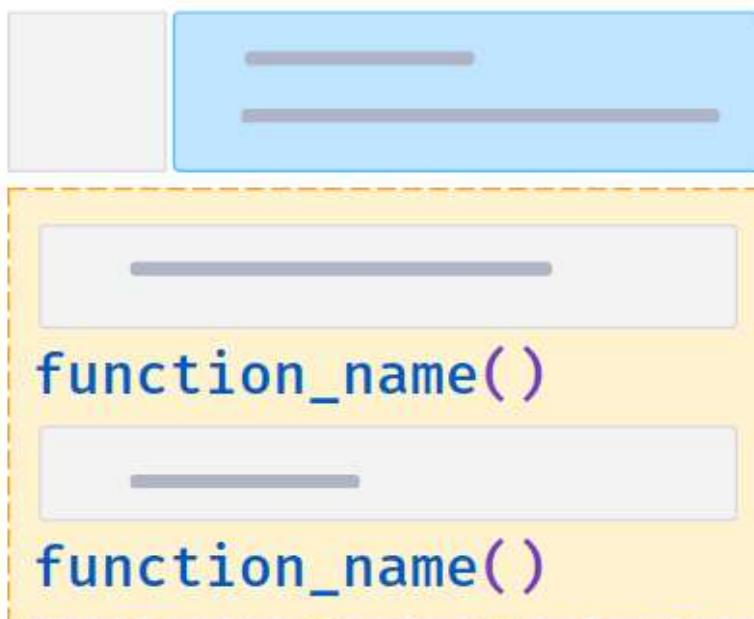
Output

Teja

### Calling a Function

The functional block of code is executed only when the function is called.

```
def function_name():
```



Code

```
1 def greet():
2     print("Hello")
3
4 name = input()
5 greet()
6 print(name)
```

### Input

```
Teja
```

### Output

```
Hello
Teja
```

## Defining & Calling a Function

A function should be defined before it is called.

### Code

```
1 name = input()
2 greet()
3 print(name)
4
5 def greet():
6     print("Hello")
```

### Input

Teja

## Output

```
NameError: name 'greet' is not defined
```

## Printing a Message

Consider the following scenario, we want to create a function, that prints a custom message, based on some variable that is defined outside the function. In the below code snippet, we want to access the value in the variable

name at line 2 in place of the ? .

## Code

PYTHON

```
1 def greet():
2     msg = "Hello " + ?
3     print(msg)
4
5 name = input()
6 greet()
```

## Input

Teja

## Desired Output

```
Hello Teja
```

We use the concept of Function Arguments for these types of scenarios.

#### Function With Arguments

We can pass values to a function using an Argument.

**def function\_name(args):**

Reusable  
Block of Code

#### Code

PYTHON

```
1 def greet(word):  
2     msg = "Hello " + word  
3     print(msg)  
4  
5 name = input()  
6 greet(word=name)
```

#### Input

Teja

#### Output

Hello Teja

## Variables Inside a Function

A variable created inside a function can only be used in it.

### Code

PYTHON

```
1 def greet(word):
2     msg = "Hello " + word
3
4 name = input()
5 greet(word=name)
6 print(msg)
```

### Input

Teja

### Output

NameError: name 'msg' is not defined

## Returning a Value

To return a value from the function use

`return` keyword.

```
def function_name(args):
```

Reusable  
Block of Code

```
return value
```

Exits from the function when return statement is executed.

Code

PYTHON

```
1 def greet(word):  
2     msg = "Hello " + word  
3     return msg  
4  
5 name = input()  
6 greeting = greet(word=name)  
7 print(greeting)
```

Input

Teja

Output

Hello Teja

Code written after

return statement will not be executed.

Code

PYTHON

```
1 def greet(word):  
2     msg = "Hello "+word  
3     return msg  
4     print(msg)  
5  
6 name = input()  
7 greeting = greet(word=name)  
8 print(greeting)
```

Input

```
Teja
```

Output

```
Hello Teja
```

### Built-in Functions

We are already using functions which are pre-defined in Python.

Built-in functions are readily available for reuse

- print()
- int()
- str()
- len()



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Function Arguments

A function can have more than one argument.

```
def function_name(arg_1, arg_2):
```

Reusable  
Block of Code

### Keyword Arguments

Passing values by their names.

#### Code

PYTHON

```
1 def greet(arg_1, arg_2):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()  
6 greet(arg_1=greeting,arg_2=name)
```

#### Input

Good Morning

Ram

## Output

Good Morning Ram

## Possible Mistakes - Keyword Arguments

### Code

PYTHON

```
1 def greet(arg_1, arg_2):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()  
6 greet(arg_2=name)
```

### Input

Good Morning

Ram

## Output

`TypeError: greet() missing 1 required positional argument: 'arg_1'`

## Positional Arguments

Values can be passed without using argument names.

- These values get assigned according to their position.
- Order of the arguments matters here.

Code

PYTHON

```
1 def greet(arg_1, arg_2):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()  
6 greet(greeting, name)
```

Input

```
Good Morning  
Ram
```

Output

```
Good Morning Ram
```

Possible Mistakes - Positional Arguments

Mistake - 1

Code

PYTHON

```
1 def greet(arg_1, arg_2):  
2     print(arg_1 + " " + arg_2)  
3
```

```
4 greeting = input()  
5 name = input()  
6 greet(greeting)
```

### Input

```
Good Morning  
Ram
```

### Output

```
TypeError: greet() missing 1 required positional argument: 'arg_2'
```

## Mistake - 2

### Code

PYTHON

```
1 def greet(arg_1, arg_2):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()  
6 greet()
```

### Input

```
Good Morning  
Ram
```

## Output

```
TypeError: greet() missing 2 required positional arguments
```

## Default Values

### Example - 1

#### Code

PYTHON

```
1 def greet(arg_1="Hi", arg_2="Ram"):
2     print(arg_1 + " " + arg_2)
3
4 greeting = input()
5 name = input()
6 greet()
```

#### Input

```
Hello
Teja
```

## Output

```
Hi Ram
```

### Example - 2

#### Code

**PYTHON**

```
1 def greet(arg_1="Hi", arg_2="Ram"):
2     print(arg_1 + " " + arg_2)
3
4 greeting = input()
5 name = input()
6 greet(greeting)
```

**Input**

```
Hello
Teja
```

**Output**

```
Hello Ram
```

*Example - 3***Code****PYTHON**

```
1 def greet(arg_1="Hi", arg_2="Ram"):
2     print(arg_1 + " " + arg_2)
3
4 greeting = input()
5 name = input()
6 greet(name)
```

**Input**

```
...
```

Hello

Teja

Output

Teja Ram

*Example - 4*

Code

PYTHON

```
1 def greet(arg_1="Hi", arg_2="Ram"):
2     print(arg_1 + " " + arg_2)
3
4 greeting = input()
5 name = input()
6 greet(arg_2=name)
```

Input

Hello

Teja

Output

Hi Teja

*Example - 5*

Code

PYTHON

```
1 def greet(arg_1="Hi", arg_2):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()  
6 greet(arg_2=name)
```

Input

```
Hello  
Teja
```

Output

```
SyntaxError:non-default argument follows default argument
```

Non-default arguments cannot follow default arguments.

*Example - 6*

Code

PYTHON

```
1 def greet(arg_2, arg_1="Hi"):  
2     print(arg_1 + " " + arg_2)  
3  
4 greeting = input()  
5 name = input()
```

```
6 greet(arg_2=name)
```

### Input

```
Hello  
Teja
```

### Output

```
Hi Teja
```

## Passing Immutable Objects

### Code

PYTHON

```
1 def increment(a):  
2     a += 1  
3  
4 a = int(input())  
5 increment(a)  
6 print(a)
```

### Input

```
5
```

### Output

Even though variable names are same, they are referring to two different objects.

Changing the value of the variable inside the function will not affect the variable outside.

 MARKED AS COMPLETE

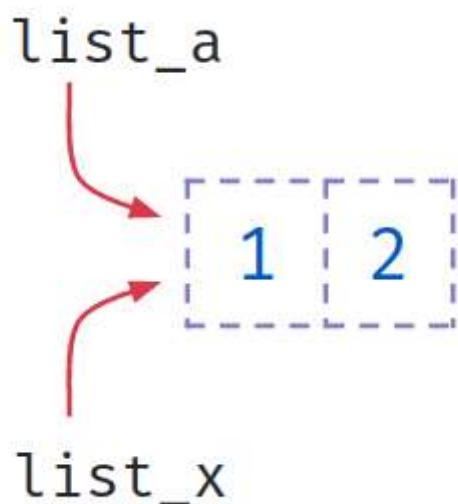
[Submit Feedback](#)

## Cheat Sheet

## Passing Mutable Objects

The same object in the memory is referred by both

list\_a and list\_x



## Code

PYTHON

```
1 def add_item(list_x):  
2     list_x += [3]  
3  
4 list_a = [1,2]  
5 add_item(list_a)  
6 print(list_a)
```

## Output

```
[1, 2, 3]
```



## Code

PYTHON

```
1 def add_item(list_x):
2     list_x = list_x + [3]
3
4 list_a = [1,2]
5 add_item(list_a)
6 print(list_a)
```

## Output

```
[1, 2]
```



Default args are evaluated only once when the function is defined, not each time the function is called.

## Code

PYTHON

```
1 def add_item(list_x=[]):
2     list_x += [3]
3     print(list_x)
4
5 add_item()
6 add_item([1,2])
7 add_item()
```

## Output

```
[3]  
[1, 2, 3]  
[3, 3]
```

## Built-in functions

Built-in functions are readily available for reuse.

We are already using functions which are pre-defined in Python

- `print()`
- `int()`
- `str()`
- `len()`

### Finding Minimum

`min()` returns the smallest item in a sequence or smallest of two or more arguments.

PYTHON

```
1 min(sequence)  
2 min(arg1, arg2, arg3 ...)
```

### Example - 1

#### Code

PYTHON

```
1 smallest= min(3,5,4)  
2 print(smallest)
```

## Output

*Example - 2*

## Code

PYTHON

```
1 smallest = min([1,-2,4,2])
2 print(smallest)
```

## Output

```
-2
```

## Minimum of Strings

```
min(str_1, str_2)
```

Strings are compared character by character using unicode values.

- P - 80(unicode)
- J - 74(unicode)

## Code

PYTHON

```
1 smallest = min("Python", "Java")
2 print(smallest)
```

## Output

Java

## Finding Maximum

`max()` returns the largest item in a sequence or largest of two or more arguments.

PYTHON

```
1 max(sequence)
2 max(arg1, arg2, arg3 ...)
```

### Example - 1

#### Code

PYTHON

```
1 largest = max(3,5,4)
2 print(largest)
```

#### Output

5

### Example - 2

#### Code

PYTHON

```
1 largest = max([1,-2,4,2])
2 print(largest)
```

## Output

4

## Finding Sum

`sum(sequence)` returns sum of items in a sequence.

## Code

PYTHON

```
1 sum_of_numbers = sum([1,-2,4,2])
2 print(sum_of_numbers)
```

## Output

5

## Ordering List Items

`sorted(sequence)` returns a new sequence with all the items in the given sequence ordered in increasing order.

## Code

PYTHON

```
1 list_a = [3, 5, 2, 1, 4, 6]
2 list_x = sorted(list_a)
3 print(list_x)
```

## Output

```
[1, 2, 3, 4, 5, 6]
```

### Ordering List Items - Reverse

`sorted(sequence, reverse=True)` returns a new sequence with all the items in the given sequence ordered in decreasing order.

#### Code

PYTHON

```
1 list_a = [3, 5, 2, 1, 4, 6]
2 list_x = sorted(list_a, reverse=True)
3 print(list_x)
```

#### Output

```
[6, 5, 4, 3, 2, 1]
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Function Call Stack & Recursion

#### Stack

Stack is a data structure that stores items in an Last-In/First-Out manner.



#### Calling a Function

#### Calling

function\_1()   **inside**   function\_2()

```
def function_1():
```

```
def function_2():
```

Code

PYTHON

```
1 def get_largest_sqr(list_x):  
2     len_list = len(list_x)  
3     for i in range(len_list):  
4         x = list_x[i]  
5         list_x[i] = x * x  
6     largest = max(list_x)  
7     return largest  
8  
9 list_a = [1,-3,2]  
10 result = get_largest_sqr(list_a)
```

```
11 print(result)
```

Collapse ^

## Output

```
9
```

In the above code calling functions are

`len()` and `max()` inside `get_largest_sqr()`

Sum of Squares of List Items

```
get_sum_of_sqrs([1,2,3])
```

## Code

PYTHON

```
1 def get_sqrd_val(x):
2     return (x * x)
3
4 def get_sum_of_sqrs(list_a):
5     sqrs_sum = 0
6     for i in list_a:
7         sqrs_sum += get_sqrd_val(i)
8     return sqrs_sum
9
10 list_a = [1, 2, 3]
11 sum_of_sqrs = get_sum_of_sqrs(list_a)
12 print(sum_of_sqrs)
```

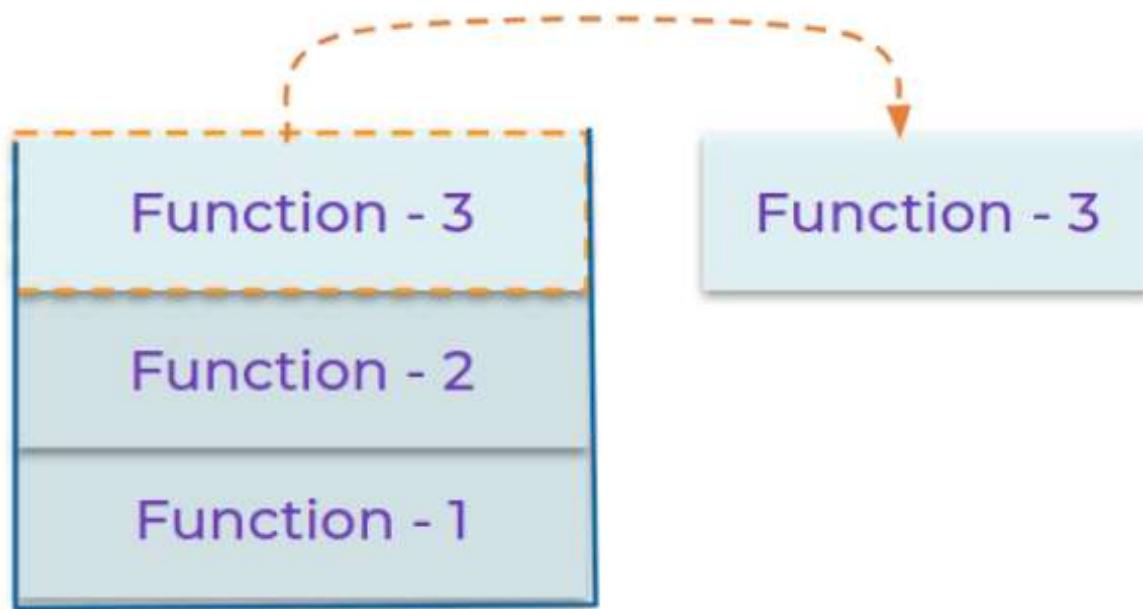
Collapse ^

## Output

```
14
```

## Function Call Stack

Function Call Stack keeps track of function calls in progress

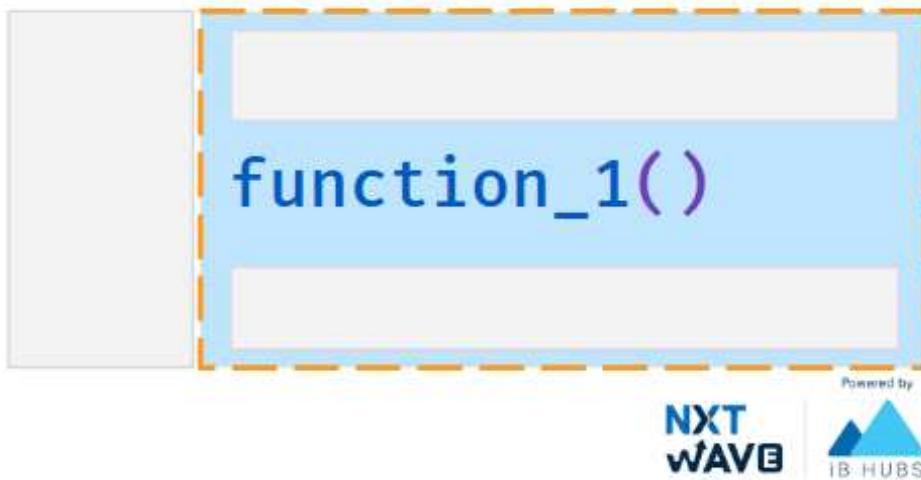


Recursion



A function calling itself is called a **Recursion**

```
def function_1():
```



Let's understand recursion with a simple example of multiplying N numbers

#### Multiply N Numbers

PYTHON

```
1 def factorial(n): # Recursive Function
2     if n == 1: # Base Case
3         return 1
4     return n * factorial(n - 1) # Recursion
5 num = int(input())
6 result = factorial(num)
7 print(result)
```

#### Base Case

A recursive function terminates when base condition is met

#### Input

3

Output

6

Without Base case

Code

PYTHON

```
1 def factorial(n):
2     return n * factorial(n - 1)
3 num = int(input())
4 result = factorial(num)
5 print(result)
```

Input

3

Output

RecursionError: maximum recursion depth exceeded

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### List Methods

Python provides `list` methods that allow us to work with lists.

Let's learn few among them

- `append()`
- `extend()`
- `insert()`
- `pop()`
- `clear()`
- `remove()`
- `sort()`
- `index()`

#### Append

`list.append(value)` Adds an element to the end of the list.

#### Code

PYTHON

```
1 list_a = []
2 for x in range(1,4):
3     list_a.append(x)
4 print(list_a)
```

#### Output

```
[1, 2, 3]
```

## Extend

`list_a.extend(list_b)` Adds all the elements of a sequence to the end of the list.

## Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_b = [4, 5, 6]
3 list_a.extend(list_b)
4 print(list_a)
```

## Output

```
[1, 2, 3, 4, 5, 6]
```

## Insert

`list.insert(index,value)` Element is inserted to the list at specified index.

## Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_a.insert(1,4)
3 print(list_a)
```

## Output

```
[1, 4, 2, 3]
```

## Pop

`list.pop()` Removes last element.

## Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_a.pop()
3 print(list_a)
```

## Output

```
[1, 2]
```

## Remove

`list.remove(value)` Removes the first matching element from the list.

## Code

PYTHON

```
1 list_a = [1, 3, 2, 3]
2 list_a.remove(3)
3 print(list_a)
```

## Output

```
[1, 2, 3]
```

## Clear

`list.clear()` Removes all the items from the list.

## Code

PYTHON

```
1 list_a = [1, 2, 3]
2 list_a.clear()
3 print(list_a)
```

## Output

```
[]
```

## Index

`list.index(value)` Returns the index at the first occurrence of the specified value.

## Code

PYTHON

```
1 list_a = [1, 3, 2, 3]
2 index = list_a.index(3)
3 print(index)
```

## Output

```
1
```

## Count

`list.count(value)` Returns the number of elements with the specified value.

### Code

PYTHON

```
1 list_a = [1, 2, 3]
2 count = list_a.count(2)
3 print(count)
```

### Output

1

## Sort

`list.sort()` Sorts the list.

### Code

PYTHON

```
1 list_a = [1, 3, 2]
2 list_a.sort()
3 print(list_a)
```

### Output

[1, 2, 3]

## Sort & Sorted

sort() Modifies the list

Code

PYTHON

```
1 list_a = [1, 3, 2]
2 list_a.sort()
3 print(list_a)
```

Output

```
[1, 2, 3]
```

sorted() Creates a new sorted list

Code

PYTHON

```
1 list_a = [1, 3, 2]
2 sorted(list_a)
3 print(list_a)
```

Output

```
[1, 3, 2]
```

✓ MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Tuples and Sequences

#### None

None is an object which is a datatype of its own (NoneType).

Used to define no value or nothing.

#### Code

PYTHON

```
1 var = None
2 print(var)
3 print(type(var))
```

#### Output

```
None
<class 'NoneType'>
```

### Function Without Return

Functions assigned to a variable, when function does not have a

return statement, the variable will get the value None

#### Code

PYTHON

```
1 def increment(a):
2     a += 1
```

```
3
4 a = 55
5 result = increment(a)
6 print(result)
```

## Output

None

## Function That Returns Nothing

When a function returns no value, the default value will be

None

### *Example - 1*

#### Code

PYTHON

```
1 def increment(a):
2     a += 1
3     return
4
5 a = 55
6 result = increment(a)
7 print(result)
```

## Output

None

*Example - 2*

Code

PYTHON

```
1 def increment(a):
2     a += 1
3     return None
4
5 a = 5
6 result = increment(a)
7 print(result)
```

Output

None

*Example - 3*

Code

PYTHON

```
1 result = print("Hi")
2 print(result)
```

Output

Hi

None

Tuple

- Holds an ordered sequence of items.
- Tuple is immutable object, where as list is a mutable object.

Code

```
1 a = 2
2 tuple_a = (5, "Six", a, 8.2)
```

PYTHON

### Creating a Tuple

- Created by enclosing elements within (round) brackets.
- Each item is separated by a comma.

Code

```
1 a = 2
2 tuple_a = (5, "Six", a, 8.2)
3 print(type(tuple_a))
4 print(tuple_a)
```

PYTHON

### Output

```
<class 'tuple'>
(5, 'Six', 2, 8.2)
```

### Tuple with a Single Item

Code

```
1 a = (1,)
2 print(type(a))
3 print(a)
```

PYTHON

## Output

```
<class 'tuple'>
(1,
```

## Accessing Tuple Elements

Accessing Tuple elements is also similar to string and list accessing and slicing.

### Code

PYTHON

```
1 a = 2
2 tuple_a = (5, "Six", a, 8.2)
3 print(tuple_a[1])
```

## Output

```
Six
```

## Tuples are Immutable

Tuples does not support modification.

### Code

PYTHON

```
1 tuple_a = (1, 2, 3, 5)
2 tuple_a[3] = 4
3 print(tuple_a)
```

## Output

```
TypeError: 'tuple' object does not support item assignment
```

## Operations can be done on Tuples

- len()
- Iterating
- Slicing
- Extended Slicing

## Converting to Tuple

`tuple(sequence)` Takes a sequence and converts it into tuple.

### String to Tuple

#### Code

PYTHON

```
1 color = "Red"
2 tuple_a = tuple(color)
3 print(tuple_a)
```

#### Output

```
('R', 'e', 'd')
```

### List to Tuple

#### Code

PYTHON

```
1 list_a = [1, 2, 3]
2 tuple_a = tuple(list_a)
3 print(tuple_a)
```

## Output

```
(1, 2, 3)
```

## Sequence to Tuple

### Code

PYTHON

```
1 tuple_a = tuple(range(4))
2 print(tuple_a)
```

## Output

```
(0, 1, 2, 3)
```

## Membership Check

Check if given data element is part of a sequence or not.

### Membership Operators

- in
- not in

### Example - 1

### Code

PYTHON

```
1 tuple_a = (1, 2, 3, 4)
2 is_part = 5 in tuple_a
3 print(is_part)
```

## Output

```
False
```

## Example - 2

### Code

PYTHON

```
1 tuple_a = (1, 2, 3, 4)
2 is_part = 1 not in tuple_a
3 print(is_part)
```

## Output

```
False
```

## List Membership

### Code

PYTHON

```
1 list_a = [1, 2, 3, 4]
2 is_part = 1 in list_a
3 print(is_part)
```

## Output

```
True
```

## String Membership

### Code

PYTHON

```
1 word = 'Python'  
2 is_part = 'th' in word  
3 print(is_part)
```

### Output

```
True
```

## Packing & Unpacking

### Unpacking

Values of any sequence can be directly assigned to variables.

Number of variables in the left should match the length of sequence.

### Code

PYTHON

```
1 tuple_a = ('R', 'e', 'd')  
2 (s_1, s_2, s_3) = tuple_a  
3 print(s_1)  
4 print(s_2)  
5 print(s_3)
```

### Output

```
R  
e  
d
```

## Errors in Unpacking

### Code

PYTHON

```
1 tuple_a = ('R', 'e', 'd')
2 s_1, s_2 = tuple_a
3 print(s_1)
4 print(s_2)
```

### Output

```
ValueError: too many values to unpack (expected 2)
```

### Code

PYTHON

```
1 tuple_a = ('R', 'e', 'd')
2 s_1, s_2, s_3, s_4 = tuple_a
3 print(s_1)
```

### Output

```
ValueError: not enough values to unpack (expected 4, got 3)
```

## Tuple Packing

0 brackets are optional while creating tuples.

In Tuple Packing, Values separated by commas will be packed into a tuple.

Code

PYTHON

```
1 a = 1, 2, 3
2 print(type(a))
3 print(a)
```

Output

```
<class 'tuple'>
(1, 2, 3)
```

Code

PYTHON

```
1 a = 1,
2 print(type(a))
3 print(a)
```

Output

```
<class 'tuple'>
(1,)
```

Code

PYTHON

```
1 a, = 1,
2 print(type(a))
3 print(a)
```

## Output

```
<class 'int'>  
1
```

 MARKED AS COMPLETE

[Submit Feedback](#)

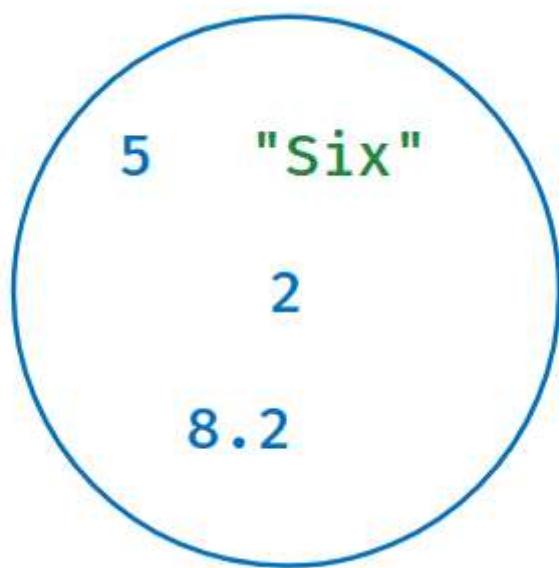
## Cheat Sheet

### Sets

Unordered collection of items.

Every set element is

- Unique (no duplicates)
- Must be immutable



### Creating a Set

- Created by enclosing elements within *{curly}* brackets.
- Each item is separated by a comma.

### Code

```
1 a = 2
2 set_a = {5, "Six", a, 8.2}
3 print(type(set_a))
4 print(set_a)
```

## Output

```
<class 'set'>
{8.2, 2, 'Six', 5}
```

Need not be in the same order as defined.

## No Duplicate Items

Sets contain unique elements

## Code

```
1 set_a = {"a", "b", "c", "a"}
2 print(set_a)
```

## Output

```
{'b', 'a', 'c'}
```

## Immutable Items

Set items must be immutable.

As List is mutable, Set cannot have list as an item.

## Code

```
1 set_a = {"a", ["c", "a"]}
2 print(set_a)
```

## Output

```
TypeError: unhashable type: 'list'
```

## Creating Empty Set

We use

`set()` to create an empty set.

## Code

```
1 set_a = set()
2 print(type(set_a))
3 print(set_a)
```

## Output

```
<class 'set'>
set()
```

## Converting to Set

`set(sequence)` takes any sequence as argument and converts to set, avoiding duplicates

### List to Set

## Code

PYTHON

```
1 set_a = set([1,2,1])
2 print(type(set_a))
3 print(set_a)
```

Output

```
<class 'set'>
{1, 2}
```

String to Set

Code

PYTHON

```
1 set_a = set("apple")
2 print(set_a)
```

Output

```
{'l', 'p', 'e', 'a'}
```

Tuple to Set

Code

PYTHON

```
1 set_a = set((1, 2, 1))
2 print(set_a)
```

Output

```
{1, 2}
```

## Accessing Items

As sets are unordered, we cannot access or change an item of a set using

- Indexing
- Slicing

### Code

PYTHON

```
1 set_a = {1, 2, 3}
2 print(set_a[1])
3 print(set_a[1:3])
```

### Output

```
TypeError: 'set' object is not subscriptable
```

## Adding Items

`set.add(value)` adds the item to the set, if the item is not present already.

### Code

PYTHON

```
1 set_a = {1, 3, 6, 2, 9}
2 set_a.add(7)
3 print(set_a)
```

### Output

```
{1, 2, 3, 6, 7, 9}
```

## Adding Multiple Items

`set.update(sequence)` adds multiple items to the set, and duplicates are avoided.

### Code

PYTHON

```
1 set_a = {1, 3, 9}
2 set_a.update([2, 3])
3 print(set_a)
```

### Output

```
{2, 1, 3, 9}
```

## Removing Specific Item

`set.discard(value)` takes a single value and removes if present.

### Code

PYTHON

```
1 set_a = {1, 3, 9}
2 set_a.discard(3)
3 print(set_a)
```

### Output

```
{1, 9}
```

`set_a.remove(value)` takes a value and remove if it present or raise an error.

Code

PYTHON

```
1 set_a = {1, 3, 9}
2 set_a.remove(5)
3 print(set_a)
```

Output

```
KeyError: 5
```

## Operations on Sets

You can perform the following operations on Sets

- `clear()`
- `len()`
- Iterating
- Membership Check



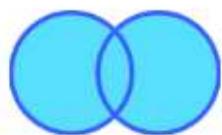
MARKED AS COMPLETE

Submit Feedback

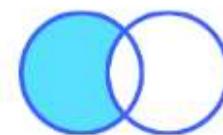
## Cheat Sheet

### Set Operations

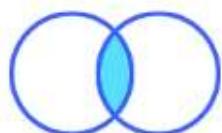
Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.



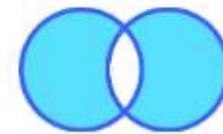
Union



Difference



Intersection



Symmetric Difference

#### Union

Union of two sets is a set containing all elements of both sets.

`set_a | set_b` or `set_a.union(sequence)`

`union()` converts sequence to a set, and performs the union.

#### Code

```
1 set_a = {4, 2, 8}  
2 set_b = {1, 2}
```

PYTHON

```
2 set_b = {1, 2}
3 union = set_a | set_b
4 print(union)
```

## Output

```
{1, 2, 4, 8}
```

## Code

PYTHON

```
1 set_a = {4, 2, 8}
2 list_a = [1, 2]
3 union = set_a.union(list_a)
4 print(union)
```

## Output

```
{1, 2, 4, 8}
```

## Intersection

Intersection of two sets is a set containing common elements of both sets.

set\_a & set\_b    or    set\_a.intersection(sequence)

intersection()    converts sequence to a set, and perform the intersection.

## Code

PYTHON

```
1 set_a = {4, 2, 8}
```

```
2 set_b = {1, 2}
3 intersection = set_a & set_b
4 print(intersection)
```

## Output

```
{2}
```

## Code

PYTHON

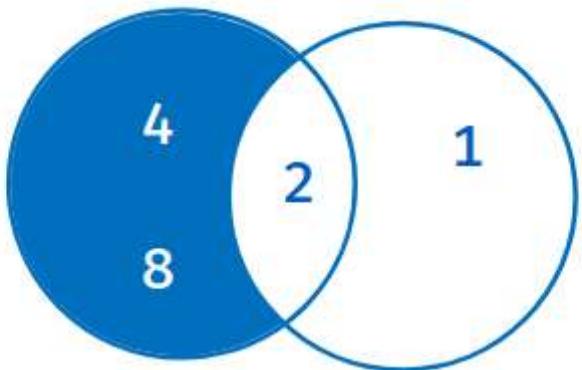
```
1 set_a = {4, 2, 8}
2 list_a = [1, 2]
3 intersection = set_a.intersection(list_a)
4 print(intersection)
```

## Output

```
{2}
```

## Difference

Difference of two sets is a set containing all the elements in the first set but not second.



```
set_a - set_b = {8, 4}
```

set\_a - set\_b or set\_a.difference(sequence)

difference() converts sequence to a set.

#### Code

PYTHON

```
1 set_a = {4, 2, 8}
2 set_b = {1, 2}
3 diff = set_a - set_b
4 print(diff)
```

#### Output

{8, 4} □

#### Code

PYTHON

```
1 set_a = {4, 2, 8}
2 tuple_a = (1, 2)
3 diff = set_a.difference(tuple_a)
4 print(diff)
```

## Output

```
{8, 4}
```

## Symmetric Difference

Symmetric difference of two sets is a set containing all elements which are not common to both sets.

`set_a ^ set_b`    or    `set_a.symmetric_difference(sequence)`

`symmetric_difference()` converts sequence to a set.

## Code

PYTHON

```
1 set_a = {4, 2, 8}
2 set_b = {1, 2}
3 symmetric_diff = set_a ^ set_b
4 print(symmetric_diff)
```

## Output

```
{8, 1, 4}
```

## Code

```
1 set_a = {4, 2, 8}
2 set_b = {1, 2}
3 diff = set_a.symmetric_difference(set_b)
4 print(diff)
```

## Output

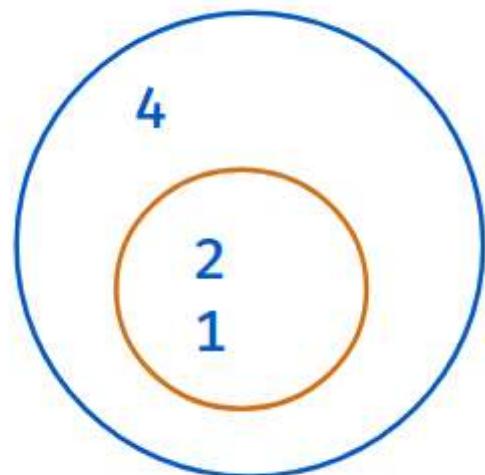
```
{8, 1, 4}
```

## Set Comparisons

Set comparisons are used to validate whether one set fully exists within another

- issubset()
- issuperset()
- isdisjoint()

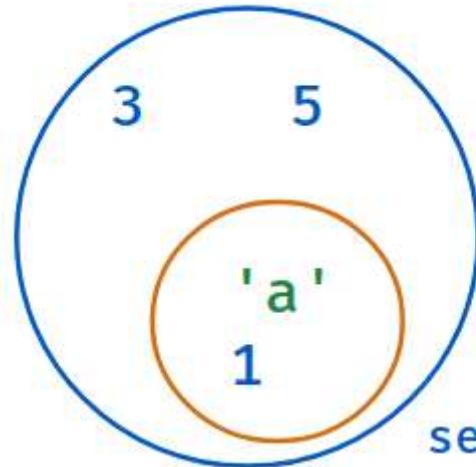
### Subset



```
set_1 = {1, 2, 4}  
set_2 = {1, 2}
```

`set2.issubset(set1)` Returns `True` if all elements of second set are in first set. Else, `False`

*Example - 1*



```
set_1 = {'a', 1, 3, 5}  
set_2 = {'a', 1}
```

True

Code

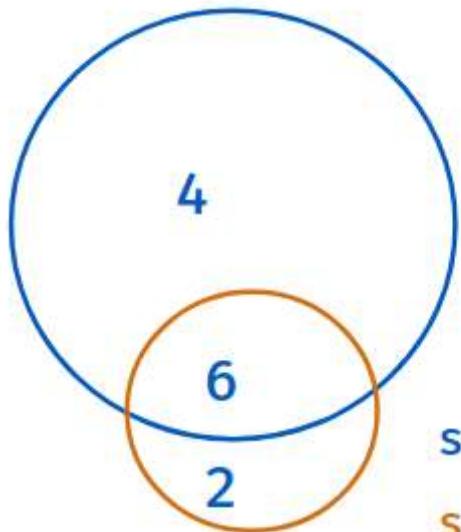
PYTHON

```
1 set_1 = {'a', 1, 3, 5}  
2 set_2 = {'a', 1}  
3 is_subset = set_2.issubset(set_1)  
4 print(is_subset)
```

Output

True

Example - 2



```
set_1 = {4, 6}  
set_2 = {2, 6}
```

**False**

Code

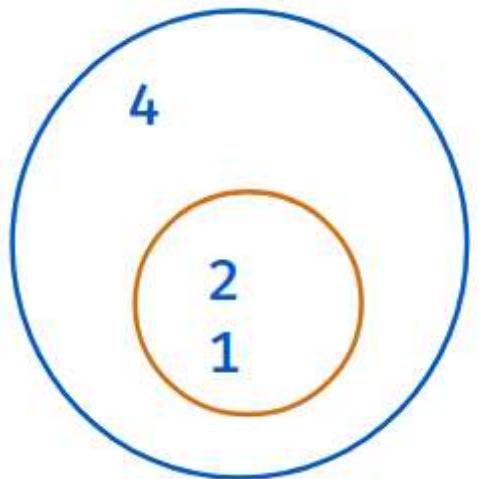
PYTHON

```
1 set_1 = {4, 6}  
2 set_2 = {2, 6}  
3 is_subset = set_2.issubset(set_1)  
4 print(is_subset)
```

Output

False

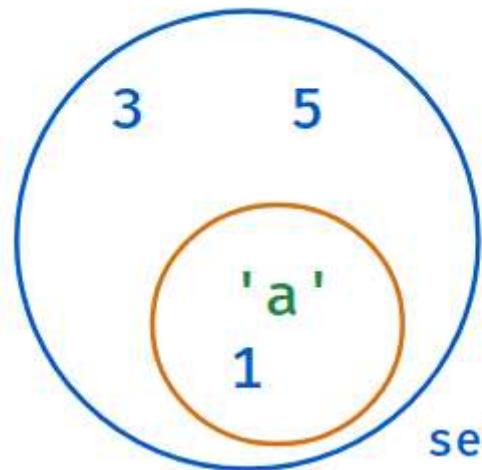
SuperSet



```
set_1 = {1, 2, 4}  
set_2 = {1, 2}
```

`set1.issuperset(set2)` Returns `True` if all elements of second set are in first set. Else, `False`

*Example - 1*



```
set_1 = {'a', 1, 3, 5}  
set_2 = {'a', 1}
```

True

Code

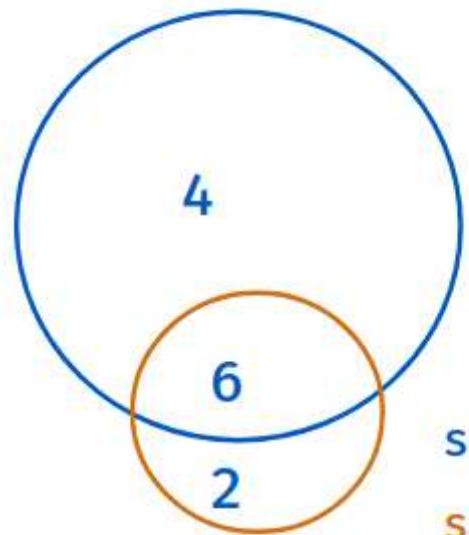
PYTHON

```
1 set_1 = {'a', 1, 3, 5}  
2 set_2 = {'a', 1}  
3 is_superset = set_1.issuperset(set_2)  
4 print(is_superset)
```

Output

True

*Example - 2*



```
set_1 = {4, 6}  
set_2 = {2, 6}
```

**False**

Code

PYTHON

```
1 set_1 = {4, 6}  
2 set_2 = {2, 6}  
3 is_superset = set_1.issuperset(set_2)  
4 print(is_superset)
```

Output

```
False
```

Disjoint Sets

`set1.isdisjoint(set2)` Returns `True` when they have no common elements. Else, `False`

Code

PYTHON

```
1 set_a = {1, 2}
2 set_b = {3, 4}
3 is_disjoint = set_a.isdisjoint(set_b)
4 print(is_disjoint)
```

Output

True

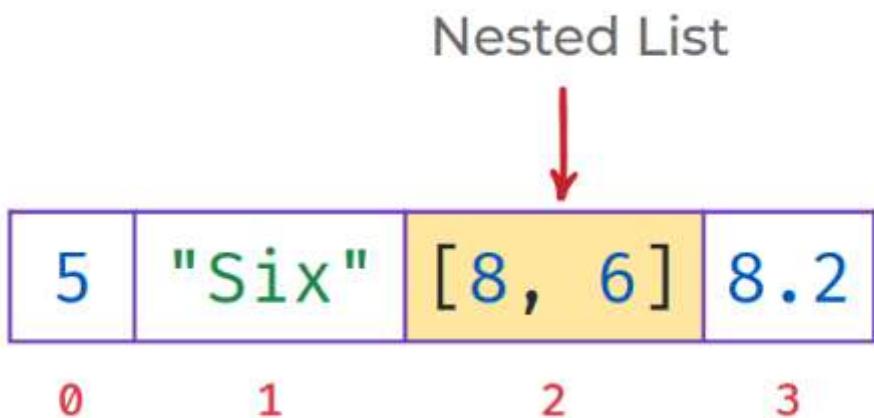
 MARKED AS COMPLETE

[Submit Feedback](#)

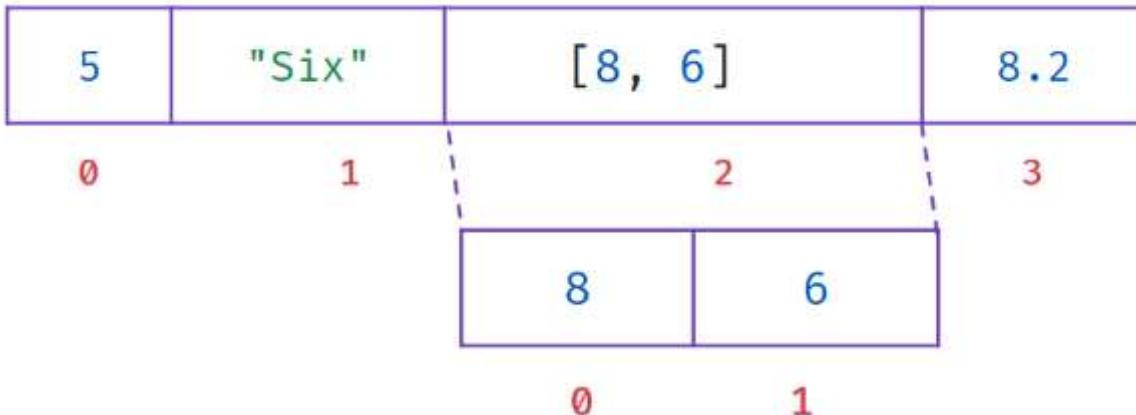
## Cheat Sheet

### Nested Lists & String Formatting

A list as an item of another list.



### Accessing Nested List



Code

PYTHON

```
1 list_a = [5, "Six", [8, 6], 8.2]
2 print(list_a[2])
```

Output

```
[8, 6]
```

Accessing Items of Nested List

*Example - 1*

Code

PYTHON

```
1 list_a = [5, "Six", [8, 6], 8.2]
2 print(list_a[2][0])
```

## Output

```
8
```

### Example - 2

#### Code

PYTHON

```
1 list_a = ["Five", "Six"]
2 print(list_a[0][1])
```

## Output

```
i
```

## String Formatting

#### Code

PYTHON

```
1 name = input()
2 age = int(input())
3 msg = ("Hi " + name + ". You are " + str(age) + " years old.")
4 print(msg)
```

String formatting simplifies this concatenation.

It increases the readability of code and type conversion is not required.

## Add Placeholders

Add placeholders

{ } where the string needs to be formatted.

PYTHON

```
1 msg = "Hi {}. You are {} years old."
2 msg.format(val_1, val_2,...)
```

Inserts values inside the string's placeholder

{ }

Code

PYTHON

```
1 name = "Raju"
2 age = 10
3 msg = "Hi {}. You are {} years old."
4 print(msg.format(name, age))
```

Output

Hi Raju. You are 10 years old.

## Number of Placeholders

Code

PYTHON

```
1 name = "Raju"
2 age = 10
3 msg = "Hi {}. You are {} years old {}."
4 print(msg.format(name, age))
```

## Output

```
IndexError: Replacement index 2 out of range for positional args tuple
```

## Numbering Placeholders

Numbering placeholders, will fill values according to the position of arguments.

### Code

PYTHON

```
1 name = input()
2 age = int(input())
3 msg = "Hi {0}. You are {1} years old."
4 print(msg.format(name, age))
```

### Input

```
Raju
```

```
10
```

## Output

```
Hi Raju. You are 10 years old.
```

### Code

PYTHON

```
1 name = input()
2 age = int(input())
3 msg = "Hi {1}. You are {0} years old."
4 print(msg.format(name, age))
```

### Input

```
Raju
10
```

### Output

```
Hi 10. You are Raju years old.
```

## Naming Placeholder

Naming placeholders will fill values according to the keyword arguments.

### Code

PYTHON

```
1 name = input()
2 age = int(input())
3 msg = "Hi {name}. You are {age} years old."
4 print(msg.format(name=name, age=age))
```

### Input

```
Raju
```

10

Output

Hi Raju. You are 10 years old.

 MARKED AS COMPLETE

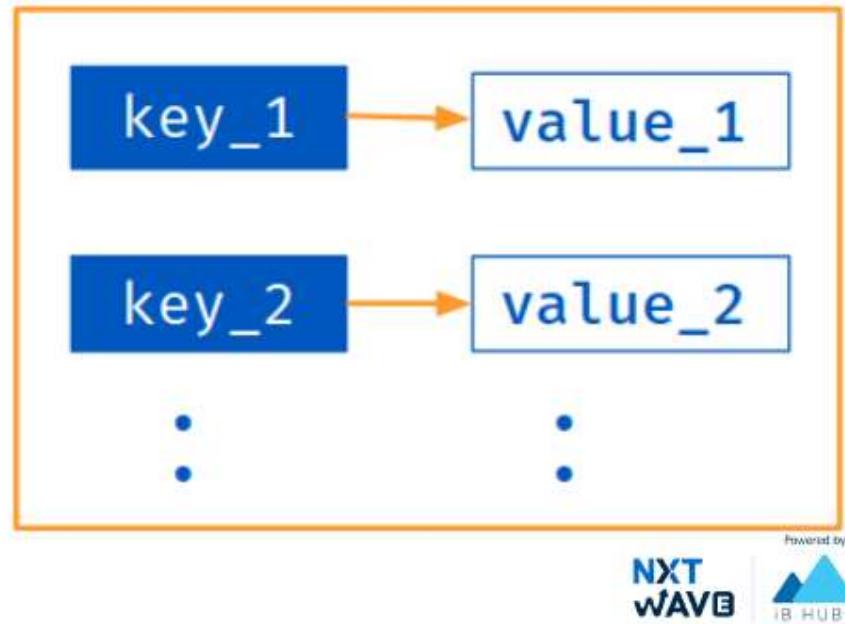
[Submit Feedback](#)

## Cheat Sheet

### Dictionaries

Unordered collection of items.

Every dictionary item is a **Key-value** pair.



### Creating a Dictionary

Created by enclosing items within **{curly}** brackets

Each item in dictionary has a key - value pair separated by a **comma**.

Code

PYTHON

```
1 dict_a = {  
2     "name": "Teja",  
3     "age": 15  
4 }
```

## Key - Value Pairs

### Code

PYTHON

```
1 dict_a = {  
2     "name": "Teja",  
3     "age": 15  
4 }
```

In the above dictionary, the

- keys are `name` and `age`
- values are `Teja` and `15`

## Collection of Key-Value Pairs

### Code

PYTHON

```
1 dict_a = { "name": "Teja",  
2             "age": 15 }  
3 print(type(dict_a))  
4 print(dict_a)
```

### Output

```
<class 'dict'>  
{'name': 'Teja', 'age': 15}
```

## Immutable Keys

Keys must be of immutable type and must be unique.

Values can be of any data type and can repeat.

Code

PYTHON

```
1 dict_a = {  
2     "name": "Teja",  
3     "age": 15,  
4     "roll_no": 15  
5 }
```

## Creating Empty Dictionary

Code - 1

PYTHON

```
1 dict_a = dict()  
2 print(type(dict_a))  
3 print(dict_a)
```

Output

```
<class 'dict'>  
{}
```

Code - 2

PYTHON

```
1 dict_a = {}  
2 print(type(dict_a))  
3 print(dict_a)
```

Output

```
<class 'dict'>  
{}
```

## Accessing Items

To access the items in dictionary, we use square bracket

[] along with the key to obtain its value.

### Code

PYTHON

```
1 def dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 print(dict_a['name'])
```

### Output

Teja

## Accessing Items - Get

The

get() method returns None if the key is not found.

### Code

PYTHON

```
1 def dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 print(dict_a.get('name'))
```

### Output

Teja

Code

PYTHON

```
1 dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 print(dict_a.get('city'))
```

Output

None

KeyError

When we use the square brackets

to access the key-value, **KeyError** is raised in case a key is not found in the dictionary.

Code

PYTHON

```
1 dict_a = {'name': 'Teja', 'age': 15 }  
2 print(dict_a['city'])
```

Output

KeyError: 'city'

Quick Tip

If we use the square brackets `[]`, `KeyError` is raised in case a key is not found in the dictionary. On the other hand, the `get()` method returns `None` if the key is not found.

### Membership Check

Checks if the given key exists.

#### Code

PYTHON

```
1 dict_a = {
2   'name': 'Teja',
3   'age': 15
4 }
5 result = 'name' in dict_a
6 print(result)
```

#### Output

True

## Operations on Dictionaries

We can update a dictionary by

- Adding a key-value pair
- Modifying existing items
- Deleting existing items

### Adding a Key-Value Pair

#### Code

PYTHON

```
1 dict_a = {'name': 'Teja', 'age': 15 }
2 dict_a['city'] = 'Goa'
```

```
3 print(dict_a)
```

## Output

```
{'name': 'Teja', 'age': 15, 'city': 'Goa'}
```

## Modifying an Existing Item

As dictionaries are mutable, we can modify the values of the keys.

### Code

PYTHON

```
1 dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 dict_a['age'] = 24  
6 print(dict_a)
```

## Output

```
{'name': 'Teja', 'age': 24}
```



## Deleting an Existing Item

We can also use the

`del` keyword to remove individual items or the entire dictionary itself.

### Code

PYTHON

```
1 dict_a = {  
2     'name': 'Teja',
```

```
3     'age': 15
4 }
5 del dict_a['age']
6 print(dict_a)
```

## Output

```
{'name': 'Teja'}
```

## Dictionary Views

They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

## Dictionary Methods

- `dict.keys()`
  - returns dictionary Keys
- `dict.values()`
  - returns dictionary Values
- `dict.items()`
  - returns dictionary items(key-value) pairs

The objects returned by

`keys()` , `values()` & `items()` are **View Objects** .

### Getting Keys

The

`keys()` method returns a view object of the type `dict_keys` that holds a list of all keys.

### Code

```
1 # dict_a = {
```

PYTHON

```
2     'name': 'Teja',
3     'age': 15
4   }
5 print(dict_a.keys())
```

## Output

```
dict_keys(['name', 'age'])
```



## Getting Values

The

`values()` method returns a view object that displays a list of all the values in the dictionary.

## Code

PYTHON

```
1 # dict_a = {
2   'name': 'Teja',
3   'age': 15
4 }
5 print(dict_a.values())
```

## Output

```
dict_values(['Teja', 15])
```

## Getting Items

The

`items()` method returns a view object that displays a list of dictionary's (key, value) tuple pairs.

## Code

PYTHON

```
1 dict_a = {
2     'name': 'Teja',
3     'age': 15
4 }
5 print(dict_a.items())
```

## Output

```
dict_items([('name', 'Teja'), ('age', 15)])
```

## Iterate over Dictionary Views

### Example - 1

## Code

PYTHON

```
1 dict_a = {
2     'name': 'Teja',
3     'age': 15
4 }
5 for key in dict_a.keys():
6     print(key)
```

## Output

```
name
age
```

### Example - 2

### Code

PYTHON

```
1 def dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 keys_list = list(dict_a.keys())  
6 print(keys_list)
```

### Output

```
['name', 'age']
```

### Example - 3

### Code

PYTHON

```
1 def dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 for value in dict_a.values():  
6     print(value)
```

### Output

```
Teja  
15
```

### Example - 4

### Code

```

1 def dict_a = {
2     'name': 'Teja',
3     'age': 15
4 }
5 for key, value in dict_a.items():
6     pair = "{} {}".format(key,value)
7     print(pair)

```

**Output**

```

name Teja
age 15

```

**Dictionary View Objects**

`keys()` , `values()` & `items()` are called Dictionary Views as they provide a dynamic view on the dictionary's items.

**Code**

```

1 def dict_a = {
2     'name': 'Teja',
3     'age': 15
4 }
5 view = dict_a.keys()
6 print(view)
7 dict_a['roll_no'] = 10
8 print(view)

```

**Output**

```

dict_keys(['name', 'age'])
dict_keys(['name', 'age', 'roll_no'])

```

## Converting to Dictionary

`dict(sequence)` takes any number of key-value pairs and converts to dictionary.

### Code

PYTHON

```
1 list_a = [
2     ("name", "Teja"),
3     ["age", 15],
4     ("roll_no", 15)
5 ]
6 dict_a = dict(list_a)
7 print(dict_a)
```

### Output

```
{'name': 'Teja', 'age': 15, 'roll_no': 15}
```

### Code

PYTHON

```
1 list_a = ["name", "Teja", 15]
2 dict_a = dict(list_a)
3 print(dict_a)
```

### Output

```
ValueError: dictionary update sequence element #0 has length 4; 2 is required
```

### Type of Keys

A dictionary key must be of a type that is immutable.

Type	Example	Can be used as key?
Integers	1000	Yes
Floats	10.25	Yes
Strings	'Hello'	Yes
Lists	[1, 5]	No (Mutable)
Sets	{'a', 'b'}	No (Mutable)
Dictionaries	{'a': 'b'}	No (Mutable)
Tuples	(1, 5)	Yes. Only if all items are immutable

Powered by



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Working with Dictionaries

#### Dictionary Methods

Python provides dictionary methods that allow us to work with dictionaries.

- `copy()`
- `get()`
- `update()`
- `fromkeys()` and more..

Let's learn few among them

#### Referring Same Dictionary Object

##### Code

PYTHON

```
1 ▶ dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 dict_b = dict_a  
6 dict_b['age'] = 20  
7 print(dict_a)  
8 print(id(dict_a))  
9 print(id(dict_b))
```

##### Output

```
{'name':'Teja', 'age': 20}
```

140170705626624

140170705626624

## Copy of Dictionary

dict.copy() returns copy of a dictionary.

### Code

PYTHON

```
1 dict_a = {  
2     'name': 'Teja',  
3     'age': 15  
4 }  
5 dict_b = dict_a.copy()  
6 dict_b['age'] = 20  
7 print(dict_a)  
8 print(id(dict_a))  
9 print(id(dict_b))
```

### Output

{'name': 'Teja', 'age': 15}

140664418952704

140664418952896

## Copy of List

### Code

PYTHON

```
1 list_a = ['Teja', 15]  
2 list_b = list_a.copy()  
3 list_b.extend([20])  
4 print(list_a)
```

```
5 print(id(list_a))
6 print(id(list_b))
```

## Output

```
['Teja', 15]
139631861316032
139631860589504
```

## Operations on Dictionaries

- [len\(\)](#)
- [clear\(\)](#)
- Membership Check

## Code

PYTHON

```
1 dict_a = {
2     'name': 'Teja',
3     'age': 15
4 }
5 print(len(dict_a)) # length of dict_a
6 if 'name' in dict_a: # Membership Check
7     print("True")
8 dict_a.clear() # clearing dict_a
9 print(dict_a)
```

## Output

```
2
True
```

```
{}
```

## Iterating

Cannot add/remove dictionary keys while iterating the dictionary.

### Code

PYTHON

```
1 dict_a = {'name': 'Teja', 'age': 15}
2 for k in dict_a.keys():
3     if k == 'name':
4         del dict_a[k]
5 print(dict_a)
```

### Output

```
RuntimeError: dictionary changed size during iteration
```

## Arbitrary Function Arguments

### Passing Multiple Values

We can define a function to receive any number of arguments.

We have already seen such functions

- `max(*args)` `max(1,2,3..)`
- `min(*args)` `min(1,2,3..)`

```
def func(*args):
```



## Variable Length Arguments

### Variable Length Arguments

Variable length arguments are packed as tuple.

#### Code

PYTHON

```
1 def more_args(*args):
2     print(args)
3
4 more_args(1, 2, 3, 4)
5 more_args()
```

#### Output

```
(1, 2, 3, 4)
()
```

### Unpacking as Arguments

If we already have the data required to pass to a function as a sequence, we can

unpack it with

- \* while passing.

## Code

PYTHON

```
1 def greet(arg1="Hi", arg2="Ram"):
2     print(arg1 + " " + arg2)
3
4 data = ["Hello", "Teja"]
5 greet(*data)
```

## Output

```
Hello Teja
```

## Multiple Keyword Arguments

We can define a function to receive any number of keyword arguments.

Variable length kwargs are packed as dictionary.

```
def func(**kwargs):
```



Variable Length  
Keyword Arguments

## Code

PYTHON

```
1 def more_args(**kwargs):
2     print(kwargs)
3
4 more_args(a=1, b=2)
5 more_args()
```

## Output

```
{'a': 1, 'b': 2}
{}
```

## Iterating

**kwargs** is a dictionary. We can iterate over them like any other dictionary.

## Code

```
1 def more_args(**kwargs):
2     for i, j in kwargs.items():
3         print('{}:{}'.format(i,j))
4
5 more_args(a=1, b=2)
```

## Output

```
a:1
b:2
```

## Unpacking as Arguments

### Code - 1

PYTHON

```
1 def greet(arg1="Hi", arg2="Ram"):
2     print(arg1 + " " + arg2)
3
4 data = {'arg1':'Hello', 'arg2':'Teja'}
5 greet(**data)
```

Output

```
Hello Teja
```

Code - 2

PYTHON

```
1 def greet(arg1="Hi", arg2="Ram"):
2     print(arg1 + " " + arg2)
3
4 data = {'msg':'Hello', 'name':'Teja'}
5 greet(**data)
```

Output

```
TypeError: greet() got an unexpected keyword argument 'msg'
```

MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Introduction to OOP

#### Good Software

Before jumping into Object Oriented Programming, let's understand the word **Software**.

Software is an easily changeable tool/product that performs a specific task.

The ease of changing or making changes to the software is referred as its softness.



- A good software should keep the users happy, by delivering what they need.
- A good software should also keep the developers happy. Ease of making changes (softness) keeps developers happy. So it should be:
  - Easy to understand and make changes.
  - Easy to fix bugs and add new features within the scope.

Object-Oriented Programming System (OOPS) is a way of *approaching, designing, developing* software that is easy to change.

### Note

Keep in mind that building software differs from solving the coding questions (that you are doing in practice).

A fundamental difference is that you move on to the next problem once you solve a coding problem. But with software, once you build it, you are often required to add new features and fix bugs that need you to make changes to the code written by you or your colleagues. So unlike the coding questions, with software, you have to keep working with the same code for a long time.

Therefore, ease of understanding (code-readability) and ease of making changes (code-maintainability) to the code become crucial in software development.

The techniques and concepts you learn in this topic are developed over time based on software developers' experiences to make working code easier.

## OOPs

Object-Oriented Programming is a way of approaching, designing and developing software, so that the components of the software and the interactions between them resembles real-life objects and their interactions.

Proper usage of OOPS concepts helps us build well-organized systems that are easy to use and extend.

### Describing Real-Life Objects

In Object Oriented Programming, we model software after real-life objects. To be good at modeling software after real-life objects, we should be able to properly describe them.

Let us try describing these real-life objects



Object 1



Object 2



The following description is a **bad way of describing**, as the information of an object scattered and unorganized.

Object 1 is a car and it has four tyres.

Object 2 is a dog and it has four legs.

Object 1 has four doors.

Object 1 can make sound.

Object 2, barks.

Object 1 is in blue color.

Object 2 is in brown color.

### Organized Description

- In the below description we are grouping the information related to an object.

Object 1 is a car and it has four tyres.

Object 1 has four doors.

Object 1 can make sound.

Object 1 is in blue color.

Object 2 is a dog and it has four legs.

Object 2, barks.

Object 2 is in brown color

- In the below approach, we further organize the information into
  - What the object is?
  - What the object has?
  - What the object can do?

Object 1 is a car

Object 1 has

Four tyres

Four seats

Four doors

and so on ...

Object 1 can

Sound horn

Move

Accelerate

and so on ...

Object 2 is a dog

Object 2 has

Brown fur

Four legs

Two ears

and so on ...

Object 2 can

Bark

Jump

Run

and so on ...

Collapse ^

The above description shows a simple framework where we describe object by specifying the properties that the object has and actions that the object can do.

Organized Description should be

- A clear separation of objects.
- A clear grouping of what object has and what it does.



MARKED AS COMPLETE

Submit Feedback

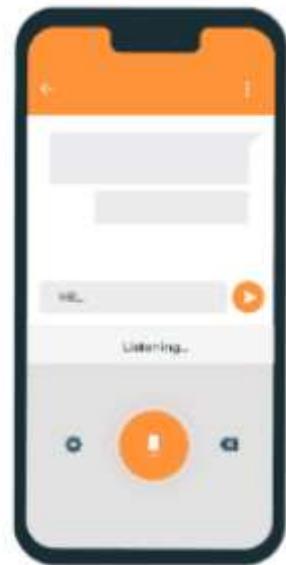
## Cheat Sheet

### Describing Similar Objects

Sometimes, the objects that we are describing are so similar that only values of the properties differ.



Object 3



Object 4



Object 3 is a Mobile

Properties

camera : 13 MP

storage : 16 GB

battery life : 21 Hrs

ram : 3 GB

and so on ...

Object 4 is a Mobile

Properties

camera : 64 MP

storage : 128 GB

battery life : 64 Hrs

ram : 6 GB

and so on ...

Collapse ^

In this case, the objects that we describe have completely the same set of properties like camera, storage, etc.

## Template

For objects that are very similar to each other (objects that have the same set of actions and properties), we can create a standard **Form** or **Template** that can be used to describe different objects in that category.

### Mobile Template

Model :

Camera:

Storage:

Does it have a Face Unlock? Yes | No

### Filled Template

Model : iPhone 12 Pro

Camera: 64MP

Storage: 128GB

Does it have a Face Unlock? Yes

## Bundling Data

While modeling real-life objects with object oriented programming, we ensure to bundle related information together to clearly separate information of different objects.

Bundling of related properties and actions together is called Encapsulation.

Classes can be used to bundle related properties and actions.



## Defining a Class

To create a class, use the keyword

```
class
```

# class ClassName:

Properties

Actions

## Special Method

In Python, a special method

`__init__` is used to assign values to properties.

## Code

PYTHON

```
1 ✎ class Mobile:  
2 ✎     def __init__(self, model, camera):  
3 ✎         self.model = model  
4 ✎         self.camera = camera
```

## Properties & Values

## Code

PYTHON

```
1 ✎ class Mobile:  
2 ✎     def __init__(self, model, camera):  
3 ✎         self.model = model  
4 ✎         self.camera = camera  
5 ✎     def make_call(self, number):  
6 ✎         print("calling..")
```

In the above example,

`model` and `camera` are the properties and values which passed to the `__init__` method.

## Action

PYTHON

```
1 ✎ class Mobile:  
2 ✎     def __init__(self, model, camera):  
3 ✎         self.model = model  
4 ✎         self.camera = camera  
5 ✎     def make_call(self, number):  
6 ✎         print("calling..")
```

In the above example, the below function is an action

PYTHON

```
1 ✎ def make_call(self, number):  
2 ✎     print("calling..")
```

In OOP terminology, properties are referred as **attributes** actions are referred as **methods**

## Using a Class

To use the defined class, we have to instantiate it.

A class is like a blueprint, while its instance is based on that class with actual values.

## Instance of Class

Syntax for creating an instance of class looks similar to function call.

An instance of class is **Object**.

## Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model, camera):  
3         self.model = model  
4         self.camera= camera  
5  
6 mobile_obj = Mobile(  
7     "iPhone 12 Pro",  
8     "12 MP")  
9 print(mobile_obj)
```

## Class Object

An object is simply a collection of attributes and methods that act on those data.

PYTHON

```
1 class Mobile:  
2     def __init__(self, model, camera):  
3         self.model = model  
4         self.camera= camera  
5  
6 mobile_obj = Mobile(  
7     "iPhone 12 Pro",  
8     "12 MP")  
9 print(mobile_obj)
```

## Method Arguments & Return Values

Similar to functions, Methods also support positional, keyword & default arguments and also return values.

### Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model, camera):  
3         self.model = model  
4         self.camera= camera  
5     def make_call(self,number):
```

```
6     |     return "calling..{}".format(number)
```

## Instance Methods of Class

For instance method, we need to first write

`self` in the function definition and then the other arguments.

### Code

PYTHON

```
1 ▶ class Mobile:  
2 ▶     def __init__(self, model, camera):  
3     |         self.model = model  
4     |         self.camera= camera  
5 ▶     def make_call(self,number):  
6     |         print("calling..{}".format(number))  
7  
8 mobile_obj = Mobile("iPhone 12 Pro", "12 MP")  
9 mobile_obj.make_call(9876543210)
```

### Output

```
calling..9876543210
```

## Multiple Instances

### Code

PYTHON

```
1 ▶ class Mobile:  
2 ▶     def __init__(self, model, camera):  
3     |         self.model = model  
4     |         self.camera= camera
```

```
5     def make_call(self,number):  
6         print("calling..{}".format(number))  
7  
8 mobile_obj1 = Mobile("iPhone 12 Pro", "12 MP")  
9 print(id(mobile_obj1))  
10 mobile_obj2 = Mobile("Galaxy M51", "64 MP")  
11 print(id(mobile_obj2))
```

Collapse ▲

## Output

```
139685004996560  
139685004996368
```

## Type of Object

The class from which object is created is considered to be the type of object.

### Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model, camera):  
3         self.model = model  
4         self.camera = camera  
5  
6 obj_1 = Mobile("iPhone 12 Pro", "12 MP")  
7 print(type(obj_1))
```

## Output

```
<class '__main__.Mobile'>
```



MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Classes and Objects

#### Attributes of an Object

Attributes can be set or accessed using

. (dot) character.

#### Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model, storage):  
3         self.model = model  
4         self.storage = storage  
5  
6  
7     obj = Mobile("iPhone 12 Pro", "128GB")  
8     print(obj.model)
```

#### Output

```
iPhone 12 Pro
```

#### Accessing in Other Methods

We can also access and update properties in other methods.

#### Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model):  
3         self.model = model  
4  
5     def get_model(self):  
6         print(self.model)  
7  
8  
9 obj_1 = Mobile("iPhone 12 Pro")  
10 obj_1.get_model()
```

## Output

```
iPhone 12 Pro
```

## Updating Attributes

It is recommended to update attributes through methods.

## Code

PYTHON

```
1 class Mobile:  
2     def __init__(self, model):  
3         self.model = model  
4  
5     def update_model(self, model):  
6         self.model = model  
7  
8  
9 obj_1 = Mobile("iPhone 12")  
10 print(obj_1.model)  
11 obj_1.update_model("iPhone 12 Pro")  
12 print(obj_1.model)
```

Collapse ^

## Output

```
iPhone 12  
iPhone 12 Pro
```

## Modeling Class

Let's model the scenario of shopping cart of ecommerce site.

The features a cart should have

- can add an item
- can remove an item from cart
- update quantity of an item
- to show list of items in cart
- to show total price for the items in the cart

## Code

PYTHON

```
1 class Cart:  
2     def __init__(self):  
3         self.items = {}  
4         self.price_details = {"book": 500, "laptop": 30000}  
5  
6     def add_item(self, item_name, quantity):  
7         self.items[item_name] = quantity  
8  
9     def remove_item(self, item_name):  
10        del self.items[item_name]  
11  
12    def update_quantity(self, item_name, quantity):  
13        self.items[item_name] = quantity  
14  
15    def get_cart_items(self):  
16        cart_items = list(self.items.keys())  
17        return cart_items
```

```
18
19     def get_total_price(self):
20         total_price = 0
21         for item, quantity in self.items.items():
22             total_price += quantity * self.price_details[item]
23         return total_price
24
25
26 cart_obj = Cart()
27 cart_obj.add_item("book", 3)
28 cart_obj.add_item("laptop", 1)
29 print(cart_obj.get_total_price())
30 cart_obj.remove_item("laptop")
31 print(cart_obj.get_cart_items())
32 cart_obj.update_quantity("book", 2)
33 print(cart_obj.get_total_price())
```

Collapse ^

## Output

```
31500
['book']
1000
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Attributes & Methods

#### Shopping Cart

- Users can add different items to their shopping cart and checkout.
- The total value of the cart should be more than a minimum amount (Rs. 100/-) for the checkout.
- During Offer Sales, all users get a flat discount on their cart and the minimum cart value will be Rs. 200/-.

#### Attributes

Broadly, attributes can be categorized as

- Instance Attributes
- Class Attributes



#### Instance Attributes

Attributes whose value can differ for each instance of class are modeled as instance attributes.

*Ex: Items in Cart*



Cart Object - A



Cart Object - B



#### Class Attributes

Attributes whose values stay common for all the objects are modelled as Class Attributes.

*Ex:* Minimum Cart Bill,

Flat Discount



Instance  
Attribute



Class  
Attributes



## Accessing Instance Attributes

### Code

```

1 class Cart:
2     flat_discount = 0
3     min_bill = 100
4     def __init__(self):
5         self.items = {}
6     def add_item(self,...):
7         self.items[item_name] = quantity
8     def display_items(self):
9         print(items)
10 a = Cart()
11 a.display_items()

```

PYTHON

Collapse ^

### Output

```
NameError: name 'items' is not defined
```

Instance attributes can only be accessed using instance of class.

## Self

`self` passed to method contains the object, which is an instance of class.

### Code

PYTHON

```
1 ▾ class Cart:  
2     flat_discount = 0  
3     min_bill = 100  
4 ▾   def __init__(self):  
5       self.items = {}  
6 ▾   def add_item(self, item_name, quantity):  
7       self.items[item_name] = quantity  
8 ▾   def display_items(self):  
9       print(self)  
10  
11 a = Cart()  
12 a.display_items()  
13 print(a)
```

Collapse ^

### Output

```
<__main__.Cart object at 0x7f6f83c9dfd0> <__main__.Cart object at 0x7f6f83c9dfd0>
```

## Accessing Using Self

### Code

PYTHON

```
1 ▾ class Cart:
```

```
2     flat_discount = 0
3     min_bill = 100
4     def __init__(self):
5         self.items = {}
6     def add_item(self, item_name, quantity):
7         self.items[item_name] = quantity
8     def display_items(self):
9         print(self.items)
10    a = Cart()
11    a.add_item("book", 3)
12    a.display_items()
```

Collapse ▲

## Output

```
{"book": 3}
```

## Accessing Using Object

### Code

PYTHON

```
1 class Cart:
2     flat_discount = 0
3     min_bill = 100
4     def __init__(self):
5         self.items = {}
6     def add_item(self, item_name, quantity):
7         self.items[item_name] = quantity
8     def display_items(self):
9         print(self.items)
10    a = Cart()
11    a.add_item("book", 3)
12    print(a.items)
```

Collapse ▲

## Output

```
{'book': 3}
```

## Accessing Using Class

### Code

PYTHON

```
1 ▶ class Cart:  
2     flat_discount = 0  
3     min_bill = 100  
4 ▶ def __init__(self):  
5     self.items = {}  
6 ▶ def add_item(self, item_name, quantity):  
7     self.items[item_name] = quantity  
8 ▶ def display_items(self):  
9     print(self.items)  
10 print(Cart.items)
```

## Output

```
AttributeError: type object 'Cart' has no attribute 'items'
```

## Accessing Class Attributes

### Example 1

### Code

PYTHON

```
1 ▶ class Cart:  
2     flat_discount = 0
```

```
3     min_bill = 100
4     def __init__(self):
5         self.items = {}
6
7 print(Cart.min_bill)
```

Output

```
100
```

Example 2

Code

PYTHON

```
1 class Cart:
2     flat_discount = 0
3     min_bill = 100
4     def __init__(self):
5         self.items = {}
6     def print_min_bill(self):
7         print(Cart.min_bill)
8
9 a = Cart()
10 a.print_min_bill()
```

Output

```
100
```

Updating Class Attribute

## Code

PYTHON

```
1 class Cart:  
2     flat_discount = 0  
3     min_bill = 100  
4     def print_min_bill(self):  
5         print(Cart.min_bill)  
6     a = Cart()  
7     b = Cart()  
8     Cart.min_bill = 200  
9     print(a.print_min_bill())  
10    print(b.print_min_bill())
```

## Output

```
200  
200
```

## Method

Broadly, methods can be categorized as

- Instance Methods
- Class Methods
- Static Methods

## Cart

```
items  
min_bill  
flat_discount  
add_item()  
display_items()  
update_flat_discount()
```

} Methods



### Instance Methods

Instance methods can access all attributes of the instance and have `self` as a parameter.

## Cart

```
items  
min_bill  
flat_discount  
add_item(self)  
display_items(self)  
update_flat_discount()
```

## Instance Methods



### Example 1

#### Code

```
1 class Cart:  
2     def __init__(self):  
3         self.items = {}  
4     def add_item(self, item_name, quantity):  
5         self.items[item_name] = quantity  
6     def display_items(self):  
7         print(self.items)  
8
```

PYTHON

```
9  a = Cart()
10 a.add_item("book", 3)
11 a.display_items()
```

Collapse ▲

## Output

```
{'book': 3}
```

## Example 2

### Code

PYTHON

```
1  class Cart:
2  def __init__(self):
3      self.items = {}
4  def add_item(self, item_name, quantity):
5      self.items[item_name] = quantity
6      self.display_items()
7  def display_items(self):
8      print(self.items)
9
10 a = Cart()
11 a.add_item("book", 3)
```

Collapse ▲

## Output

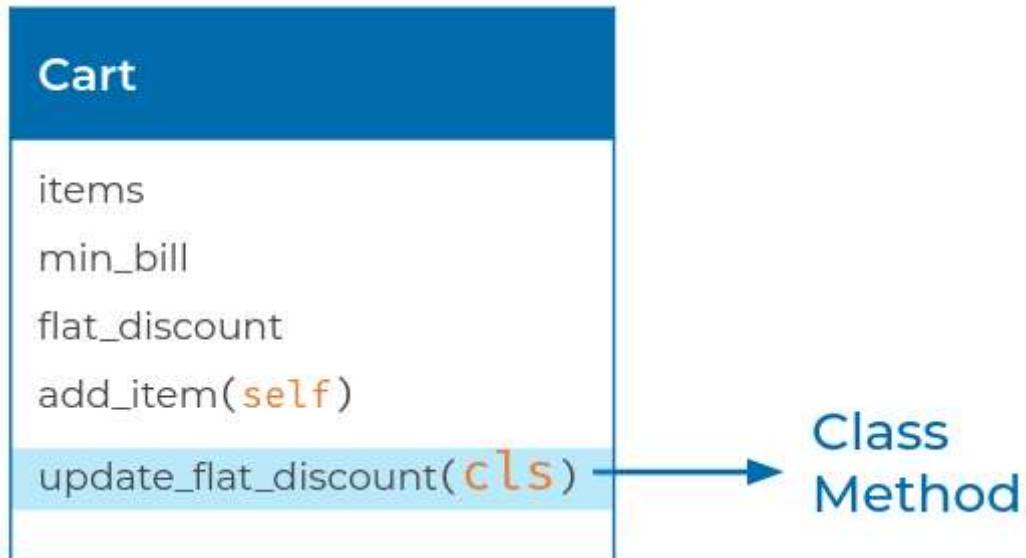
```
{'book': 3}
```

## Class Methods

Methods which need access to class attributes but not instance attributes are marked as Class Methods.

For class methods, we send

`cls` as a parameter indicating we are passing the class.



Code

PYTHON

```
1 class Cart:  
2     flat_discount = 0  
3     min_bill = 100  
4     @classmethod  
5     def update_flat_discount(cls,  
6         new_flat_discount):  
7         cls.flat_discount = new_flat_discount
```

```
8  
9  Cart.update_flat_discount(25)  
10 print(Cart.flat_discount)
```

## Output

```
25
```

`@classmethod` decorator marks the method below it as a class method.

We will learn more about decorators in upcoming sessions.

## Accessing Class Method

### Code

PYTHON

```
1 class Cart:  
2     flat_discount = 0  
3     min_bill = 100  
4     @classmethod  
5     def update_flat_discount(cls, new_flat_discount):  
6         cls.flat_discount = new_flat_discount  
7  
8     @classmethod  
9     def increase_flat_discount(cls, amount):  
10        new_flat_discount = cls.flat_discount + amount  
11        cls.update_flat_discount(new_flat_discount)  
12  
13    Cart.increase_flat_discount(50)  
14    print(Cart.flat_discount)
```

Collapse ▲

## Output

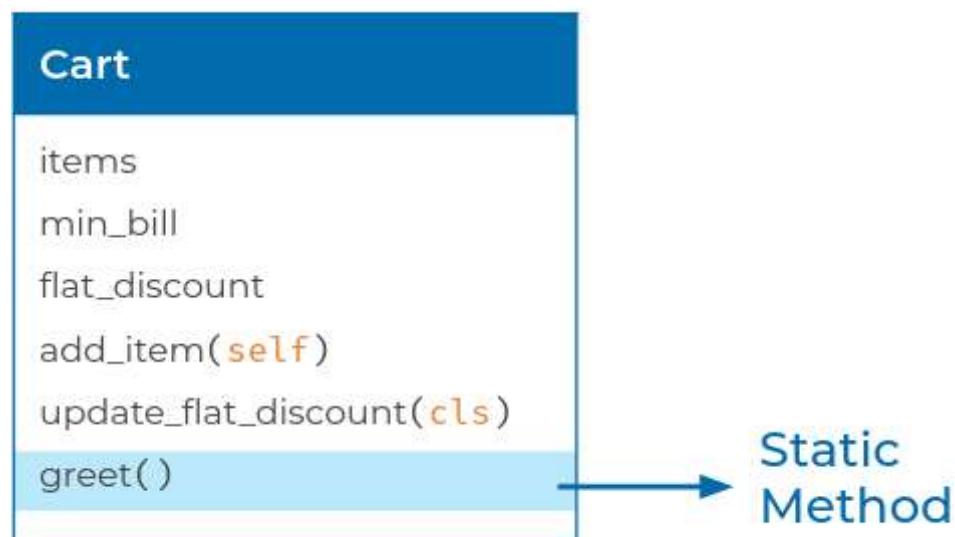
## Static Method

We might need some generic methods that don't need access to either instance or class attributes. These type of methods are called Static Methods.

Usually, static methods are used to create utility functions which make more sense to be part of the class.

`@staticmethod` decorator marks the method below it as a static method.

We will learn more about decorators in upcoming sessions.



```
1 class Cart:  
2  
3     @staticmethod  
4     def greet():  
5         print("Have a Great Shopping")  
6  
7 Cart.greet()
```

## Output

```
Have a Great Shopping
```

## Overview of Instance, Class & Static Methods

Instance Methods	Class Methods	Static Methods
self as parameter	cls as parameter	No cls or self as parameters
No decorator required	Need decorator @classmethod	Need decorator @staticmethod
Can be accessed through object(instance of class)	Can be accessed through class	Can be accessed through class

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Inheritance

#### Products

Lets model e-commerce site having different products like Electronics, Kids Wear, Grocery, etc.

#### Electronic Item

Following are few attributes & methods for an Electronic product.

## Electronic Item

Name

Price

Deal Price

Rating

Warranty in Months

Specifications

display\_product\_details(**self**)

get\_warranty(**self**)

and more ...



## Grocery Item

Similarly, attribute & methods for a Grocery item.

## Grocery Item

Name

Price

Deal Price

Rating

Packed Date

Expiry Date

display\_product\_details(**self**)

get\_expiry\_date(**self**)



### Common Attributes & Methods

All these products Electronics, Kids Wear, Grocery etc.. have few common attributes & methods.

Electronic Item	Grocery Item
Name	
Price	
Deal Price	
Rating	
Warranty in Months	Packed Date
Specifications	Expiry Date
display_product_details( <code>self</code> )	display_product_details( <code>self</code> )
get_warranty( <code>self</code> )	get_expiry_date( <code>self</code> )

#### Specific Attributes & Methods

Also, each product has specific attributes & methods of its own.



Electronic Item	Grocery Item
Name	Name
Price	Price
Deal Price	Deal Price
Rating	Rating
Warranty in Months	Packed Date
Specifications	Expiry Date
display_product_details( <b>self</b> )	display_product_details( <b>self</b> )
get_warranty( <b>self</b> )	get_expiry_date( <b>self</b> )



## Electronic & Grocery Items

Electronic Item & Grocery Item will have all attributes & methods which are common to all products.

Lets Separate the common attributes & methods as Product

## Product

Name  
Price  
Deal Price  
Rating  
display\_product\_details(**self**)

## Electronic Item

Name  
Price  
Deal Price  
Rating  
display\_product\_details(**self**)

Warranty in Months  
Specifications  
get\_warranty(**self**)

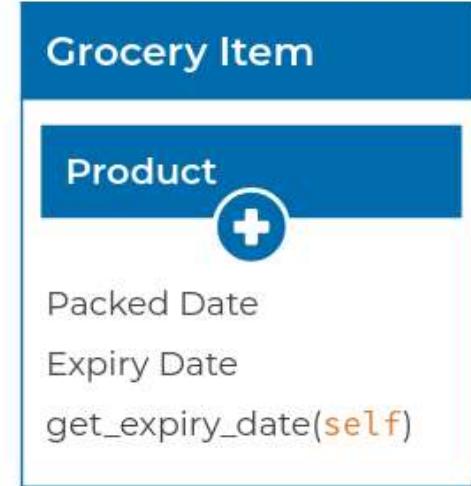
## Grocery Item

Name  
Price  
Deal Price  
Rating  
display\_product\_details(**self**)

Packed Date  
Expiry Date  
get\_expiry\_date(**self**)

Modelling Classes





#### Advantages of Modelling Classes as above

- Reusability
- Clear Separation
- More Organized

#### Inheritance

Inheritance is a mechanism by which a class inherits attributes and methods from another class.

With Inheritance, we can have

ElectronicItem inherit the attributes & methods from Product instead of defining them again.

Product is Super/Base/Parent Class and ElectronicItem is Sub/Derived/Child Class.

## Electronic Item

### Product



Warranty in Months

Specifications

get\_warranty(**self**)

## Grocery Item

### Product



Packed Date

Expiry Date

get\_expiry\_date(**self**)

Super Class

Code

PYTHON

```
1 ▾ class Product:  
2 ▾     def __init__(self, name, price, deal_price, ratings):  
3         self.name = name  
4         self.price = price  
5         self.deal_price = deal_price  
6         self.ratings = ratings  
7         self.you_save = price - deal_price  
8 ▾     def display_product_details(self):  
9         print("Product: {}".format(self.name))  
10        print("Price: {}".format(self.price))  
11        print("Deal Price: {}".format(self.deal_price))
```



```
12     print("You Saved: {}".format(self.you_save))
13     print("Ratings: {}".format(self.ratings))
14
15 p = Product("Shoes",500, 250, 3.5)
16 p.display_product_details()
```

Collapse ▲

## Output

```
Product: Shoes Price: 500 Deal Price: 250 You Saved: 250 Ratings: 3.5
```

## Sub Class

The subclass automatically inherits all the attributes & methods from its superclass.

### Example 1

#### Code

PYTHON

```
1 class Product:
2     def __init__(self, name, price, deal_price, ratings):
3         self.name = name
4         self.price = price
5         self.deal_price = deal_price
6         self.ratings = ratings
7         self.you_save = price - deal_price
8     def display_product_details(self):
9         print("Product: {}".format(self.name))
10        print("Price: {}".format(self.price))
11        print("Deal Price: {}".format(self.deal_price))
12        print("You Saved: {}".format(self.you_save))
13        print("Ratings: {}".format(self.ratings))
14
15 class ElectronicItem(Product):
16     pass
17 class GroceryItem(Product):
18     pass
```

```
19  
20 e = ElectronicItem("TV", 45000, 40000, 3.5)  
21 e.display_product_details()
```

Collapse ▲

## Output

```
Product: TV Price: 45000 Deal Price: 40000 You Saved: 5000 Ratings: 3.5
```

## Example 2

### Code

PYTHON

```
1 class Product:  
2     def __init__(self, name, price, deal_price, ratings):  
3         self.name = name  
4         self.price = price  
5         self.deal_price = deal_price  
6         self.ratings = ratings  
7         self.you_save = price - deal_price  
8     def display_product_details(self):  
9         print("Product: {}".format(self.name))  
10        print("Price: {}".format(self.price))  
11        print("Deal Price: {}".format(self.deal_price))  
12        print("You Saved: {}".format(self.you_save))  
13        print("Ratings: {}".format(self.ratings))  
14  
15 class ElectronicItem(Product):  
16     pass  
17 class GroceryItem(Product):  
18     pass  
19  
20 e = GroceryItem("milk", 25, 20, 3)  
21 e.display_product_details()
```

Collapse ▲

## Output

```
Product: milk Price: 25 Deal Price: 20 You Saved: 5 Ratings: 3
```

### Example 3

#### Code

PYTHON

```
1 class Product:
2     def __init__(self, name, price, deal_price, ratings):
3         self.name = name
4         self.price = price
5         self.deal_price = deal_price
6         self.ratings = ratings
7         self.you_save = price - deal_price
8     def display_product_details(self):
9         print("Product: {}".format(self.name))
10        print("Price: {}".format(self.price))
11        print("Deal Price: {}".format(self.deal_price))
12        print("You Saved: {}".format(self.you_save))
13        print("Ratings: {}".format(self.ratings))
14
15 class ElectronicItem(Product):
16     def set_warranty(self, warranty_in_months):
17         self.warranty_in_months = warranty_in_months
18
19     def get_warranty(self):
20         return self.warranty_in_months
21
22 e = ElectronicItem("TV", 45000, 40000, 3.5)
23 e.set_warranty(24)
24 print(e.get_warranty())
```

Collapse ▲

#### Output

In the above example, calling

`set_warranty` will create an attribute `warranty_in_months`.

### Super Class & Sub Class

Superclass cannot access the methods and attributes of the subclass.

### Code

PYTHON

```
1  class Product:
2      def __init__(self, name, price, deal_price, ratings):
3          self.name = name
4          self.price = price
5          self.deal_price = deal_price
6          self.ratings = ratings
7          self.you_save = price - deal_price
8      def display_product_details(self):
9          print("Product: {}".format(self.name))
10         print("Price: {}".format(self.price))
11         print("Deal Price: {}".format(self.deal_price))
12         print("You Saved: {}".format(self.you_save))
13         print("Ratings: {}".format(self.ratings))
14
15 class ElectronicItem(Product):
16     def set_warranty(self, warranty_in_months):
17         self.warranty_in_months = warranty_in_months
18
19     def get_warranty(self):
20         return self.warranty_in_months
21
22 p = Product("TV", 45000, 40000, 3.5)
23 p.set_warranty(24)
```

Collapse ▲

## Output

```
AttributeError: 'Product' object has no attribute 'set_warranty'
```

## Sub Class Method

### Code

PYTHON

```
1 class Product:
2     def __init__(self, name, price, deal_price, ratings):
3         self.name = name
4         self.price = price
5         self.deal_price = deal_price
6         self.ratings = ratings
7         self.you_save = price - deal_price
8     def display_product_details(self):
9         print("Product: {}".format(self.name))
10        print("Price: {}".format(self.price))
11        print("Deal Price: {}".format(self.deal_price))
12        print("You Saved: {}".format(self.you_save))
13        print("Ratings: {}".format(self.ratings))
14
15 class ElectronicItem(Product):
16     def set_warranty(self, warranty_in_months):
17         self.warranty_in_months = warranty_in_months
18
19     def get_warranty(self):
20         return self.warranty_in_months
21
22 e = ElectronicItem("TV", 45000, 40000, 3.5)
23 e.set_warranty(24)
24 e.display_product_details()
```

Collapse ▲

## Output

```
Product: TV Price: 45000 Deal Price: 40000 You Saved: 5000 Ratings: 3.5
```

## Calling Super Class Method

We can call methods defined in superclass from the methods in the subclass.

## Code

PYTHON

```
1 class Product:
2     def __init__(self, name, price, deal_price, ratings):
3         self.name = name
4         self.price = price
5         self.deal_price = deal_price
6         self.ratings = ratings
7         self.you_save = price - deal_price
8     def display_product_details(self):
9         print("Product: {}".format(self.name))
10        print("Price: {}".format(self.price))
11        print("Deal Price: {}".format(self.deal_price))
12        print("You Saved: {}".format(self.you_save))
13        print("Ratings: {}".format(self.ratings))
14
15 class ElectronicItem(Product):
16     def set_warranty(self, warranty_in_months):
17         self.warranty_in_months = warranty_in_months
18
19     def get_warranty(self):
20         return self.warranty_in_months
21
22     def display_electronic_product_details(self):
23         self.display_product_details()
24         print("Warranty {} months".format(self.warranty_in_months))
25
26 e = ElectronicItem("TV", 45000, 40000, 3.5)
27 e.set_warranty(24)
```

```
27 e.set_warranty(24)
28 e.display_electronic_product_details()
```

Collapse ▲

## Output

```
Product: TV Price: 45000 Deal Price: 40000 You Saved: 5000 Ratings: 3.5 Warranty 24 months
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## Cheat Sheet

### Inheritance - Part 2

**How would you design and implement placing order with the details of all the products bought?**

#### Composition

Modelling instances of one class as attributes of another class is called **Composition**

#### Code

PYTHON

```
1 ▶ class Product:
2
3 ▶     def __init__(self, name, price, deal_price, ratings):
4         self.name = name
5         self.price = price
6         self.deal_price = deal_price
7         self.ratings = ratings
8         self.you_save = price - deal_price
9
10    def display_product_details(self):
11        print("Product: {}".format(self.name))
12        print("Price: {}".format(self.price))
13        print("Deal Price: {}".format(self.deal_price))
14        print("You Saved: {}".format(self.you_save))
15        print("Ratings: {}".format(self.ratings))
16
17    def get_deal_price(self):
18        return self.deal_price
19
20 ▶ class ElectronicItem(Product):
21    def set_warranty(self, warranty_in_months):
22        self.warranty_in_months = warranty_in_months
```

```
22         self.warranty_in_months = warranty_in_months
23
24     def get_warranty(self):
25         return self.warranty_in_months
26
27 class GroceryItem(Product):
28     pass
29
30 class Order:
31     def __init__(self, delivery_speed, delivery_address):
32         self.items_in_cart = []
33         self.delivery_speed = delivery_speed
34         self.delivery_address = delivery_address
35
36     def add_item(self, product, quantity):
37         self.items_in_cart.append((product, quantity))
38
39     def display_order_details(self):
40         for product, quantity in self.items_in_cart:
41             product.display_product_details()
42         print("Quantity: {}".format(quantity))
43
44     def display_total_bill(self):
45         total_bill = 0
46         for product, quantity in self.items_in_cart:
47             price = product.get_deal_price() * quantity
48             total_bill += price
49         print("Total Bill: {}".format(total_bill))
50
51 milk = GroceryItem("Milk", 40, 25, 3.5)
52 tv = ElectronicItem("TV", 45000, 40000, 3.5)
53 order = Order("Prime Delivery", "Hyderabad")
54 order.add_item(milk, 2)
55 order.add_item(tv, 1)
56 order.display_order_details()
57 order.display_total_bill()
```

Collapse ▲

## Output

```
Product: Milk
Price: 40
Deal Price: 25
You Saved: 15
Ratings: 3.5
Quantity: 2
Product: TV
Price: 45000
Deal Price: 40000
You Saved: 5000
Ratings: 3.5
Quantity: 1
Total Bill: 40050
```

[Collapse ^](#)

In the above example, we are modelling **Product** as attribute of **Order**

## Overriding Methods

Sometimes, we require a method in the instances of a sub class to behave differently from the method in instance of a superclass.

## Code

PYTHON

```
1 ▶ class Product:
2
3 ▶     def __init__(self, name, price, deal_price, ratings):
4         self.name = name
5         self.price = price
6         self.deal_price = deal_price
7         self.ratings = ratings
8         self.you_save = price - deal_price
9
10 ▶    def display_product_details(self):
11        print("Product Name: ", self.name)
12        print("Original Price: ", self.price)
13        print("Deal Price: ", self.deal_price)
14        print("You Saved: ", self.you_save)
15        print("Ratings: ", self.ratings)
```

```
11     print("Product: {}".format(self.name))
12     print("Price: {}".format(self.price))
13     print("Deal Price: {}".format(self.deal_price))
14     print("You Saved: {}".format(self.you_save))
15     print("Ratings: {}".format(self.ratings))
16
17 def get_deal_price(self):
18     return self.deal_price
19
20 class ElectronicItem(Product):
21
22 def display_product_details(self):
23     self.display_product_details()
24     print("Warranty {} months".format(self.warranty_in_months))
25
26 def set_warranty(self, warranty_in_months):
27     self.warranty_in_months = warranty_in_months
28
29 def get_warranty(self):
30     return self.warranty_in_months
31
32 e = ElectronicItem("Laptop", 45000, 40000, 3.5)
33 e.set_warranty(10)
34 e.display_product_details()
```

Collapse ▲

## Output

```
RecursionError: maximum recursion depth exceeded
```

Because

self.display\_product\_details() in ElectronicItem class does not call the method in the superclass.

## Super

## Accessing Super Class's Method

`super()` allows us to call methods of the superclass (`Product`) from the subclass.

Instead of writing and methods to access and modify warranty we can override

`_init_`

Let's add warranty of `ElectronicItem`.

Code

PYTHON

```
1  class Product:
2
3      def __init__(self, name, price, deal_price, ratings):
4              self.name = name
5              self.price = price
6              self.deal_price = deal_price
7              self.ratings = ratings
8              self.you_save = price - deal_price
9
10     def display_product_details(self):
11             print("Product: {}".format(self.name))
12             print("Price: {}".format(self.price))
13             print("Deal Price: {}".format(self.deal_price))
14             print("You Saved: {}".format(self.you_save))
15             print("Ratings: {}".format(self.ratings))
16
17     def get_deal_price(self):
18             return self.deal_price
19
20 class ElectronicItem(Product):
21
22     def display_product_details(self):
23             super().display_product_details()
24             print("Warranty {} months".format(self.warranty_in_months))
25
26     def set_warranty(self, warranty_in_months):
```

```
26     def set_warranty(self, warranty_in_months):
27         self.warranty_in_months = warranty_in_months
28
29     def get_warranty(self):
30         return self.warranty_in_months
31
32 e = ElectronicItem("Laptop",45000, 40000,3.5)
33 e.set_warranty(10)
34 e.display_product_details()
```

Collapse ^

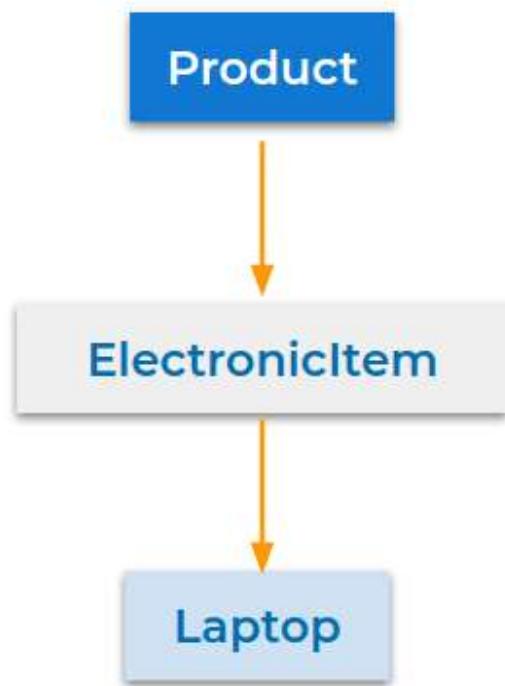
## Output

```
Product: Laptop
Price: 45000
Deal Price: 40000
You Saved: 5000
Ratings: 3.5
Warranty 10 months
```

## MultiLevel Inheritance

We can also inherit from a subclass. This is called **MultiLevel Inheritance**.

We can continue such inheritance to any depth in Python.



PYTHON

```
1 class Product:  
2     pass  
3  
4 class ElectronicItem(Product):  
5     pass  
6  
7 class Laptop(ElectronicItem):  
8     pass
```

## When to use Inheritance?

Prefer modeling with inheritance when the classes have an **IS-A** relationship.

Car is a Vehicle ✓

Truck is a Vehicle ✓

Car is a Tyre ✗

Order is a Product ✗



## When to use Composition?

Prefer modeling with inheritance when the classes have an **HAS-A** relationship.

Car has a Tyre ✓

Order has a Product ✓

Product has a Order ✗

Car has a Vehicle ✗



✓ MARKED AS COMPLETE

Submit Feedback

## 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

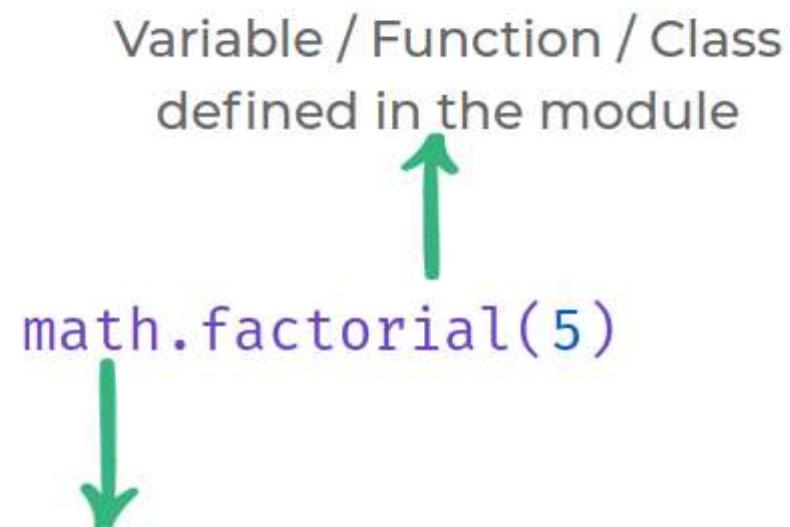
To use a functionality defined in a module we need to import that module in our program.

PYTHON

```
1 import module_name
```

### Math Module

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



## Code

PYTHON

```
1 import math  
2 print(math.factorial(5))  
3 print(math.pi)
```

## Output

```
120  
3.141592653589793
```

## Importing module

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

## Code

PYTHON

```
1 import math as m1  
2 print(m1.factorial(5))
```

## Output

```
120
```

## Importing from a Module

We can import just a specific definition from a module.

## Code

PYTHON

```
1 from math import factorial
```

```
2 print(factorial(5))
```

## Output

```
120
```

## Aliasing Imports

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

## Code

PYTHON

```
1 from math import factorial as fact
2 print(fact(5))
```

## 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

PYTHON

```
1 import random  
2 random_integer = random.randint(1, 10)  
3 print(random_integer)
```

## Output

## Choice

`choice()` is a function in random module which returns a random element from the sequence.

## Code

PYTHON

```
1 import random  
2 random_ele = random.choice(["A", "B", "C"])  
3 print(random_ele)
```

## 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

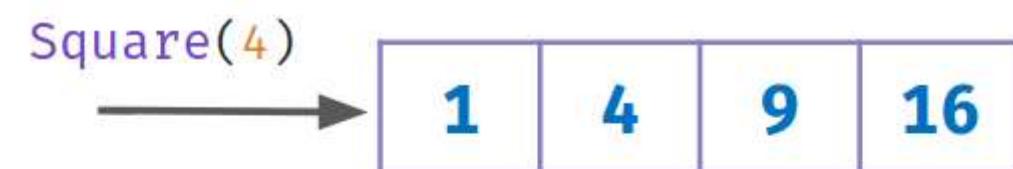
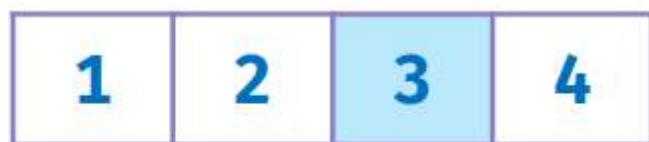
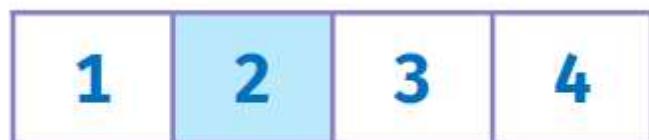
We worked with different sequences (list, tuples, etc.)

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 def square(n):  
2     return n * n  
3 numbers = [1, 2, 3, 4]  
4 result = map(square, numbers)  
5 print(list(result))
```

PYTHON

```
5 numbers_square = list(result)
6 print(numbers_square)
```

## Output

```
[1, 4, 9, 16]
```

## Example - 2

### Code

PYTHON

```
1 numbers = list(map(int, input().split()))
2 print(numbers)
```

### Input

```
1 2 3 4
```

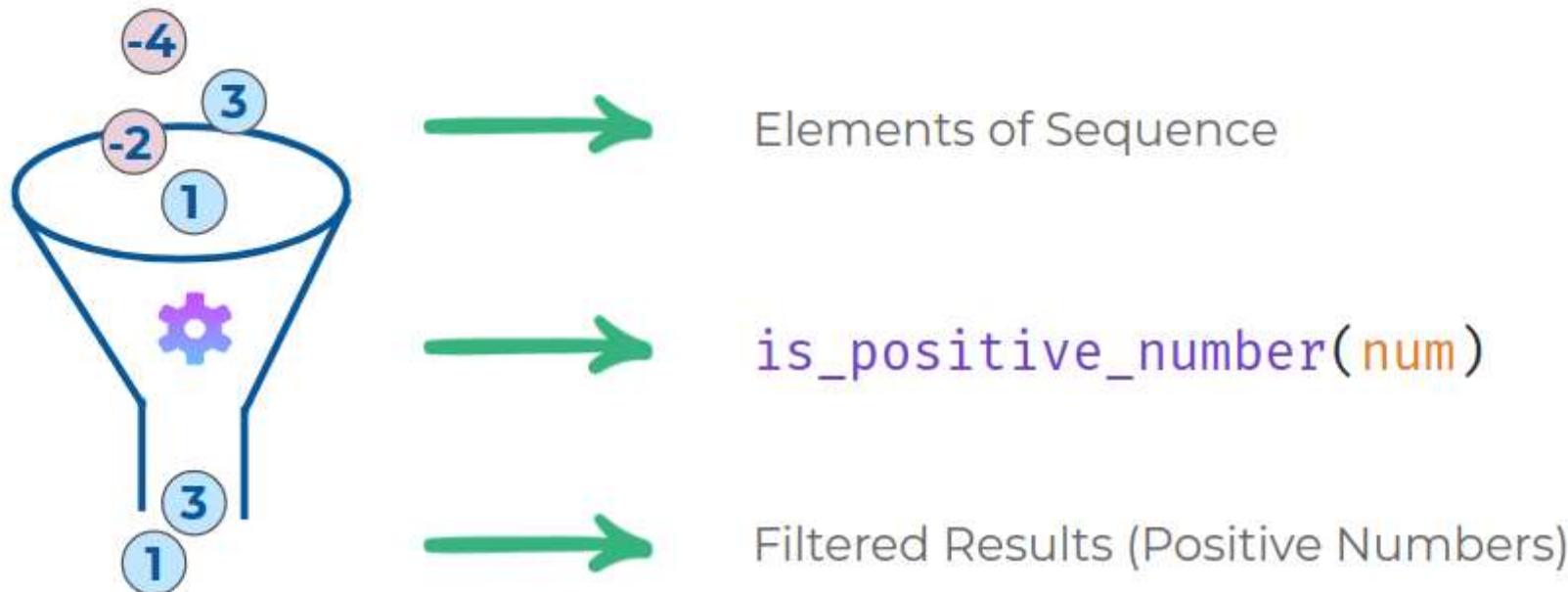
## Output

```
[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 def is_positive_number(num):  
2     return num > 0  
3  
4 list_a = [1, -2, 3, -4]  
5 positive_nums = filter(is_positive_number, list_a)  
6 print(list(positive_nums))
```

PYTHON

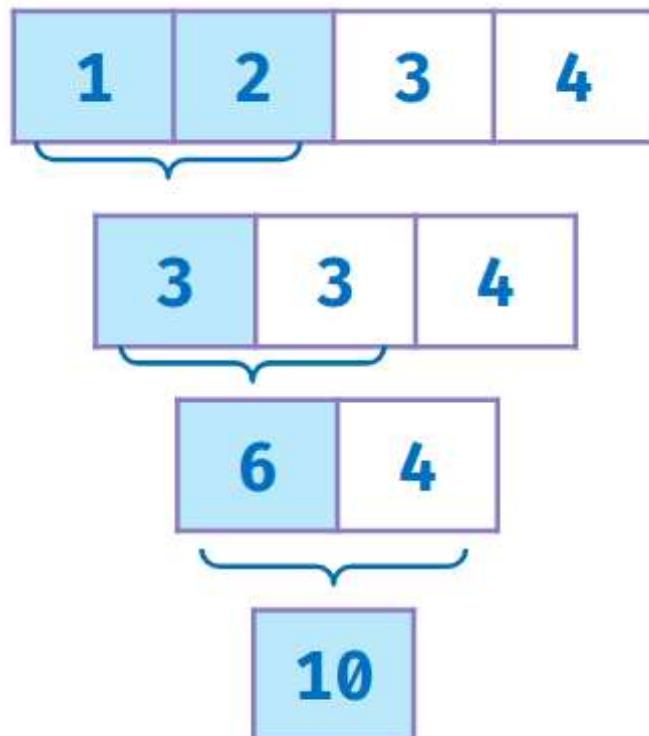
## Output

[1, 3]

U

## Reduce

reduce() function is defined in the functools module.



## Code

PYTHON

```
1  from functools import reduce  
2  
3  def sum_of_num(a, b):
```

```
4     return a+b
5
6 list_a = [1, 2, 3, 4]
7 sum_of_list = reduce(sum_of_num, list_a)
8 print(sum_of_list)
```

### Output

```
10
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## 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.

"A"

1.25



#### 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.

"A"

Id - 140035229724336



Id - 139630925071104



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

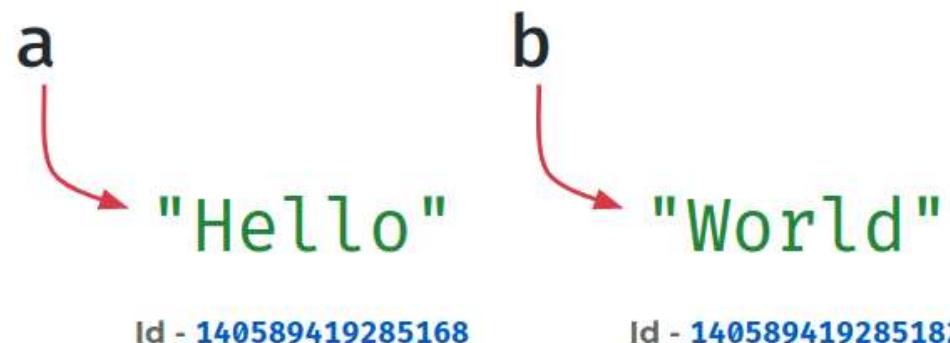
The unique id will be related to the location where the object is stored in the **Computer Memory**.

**Name of an Object**

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

### Code

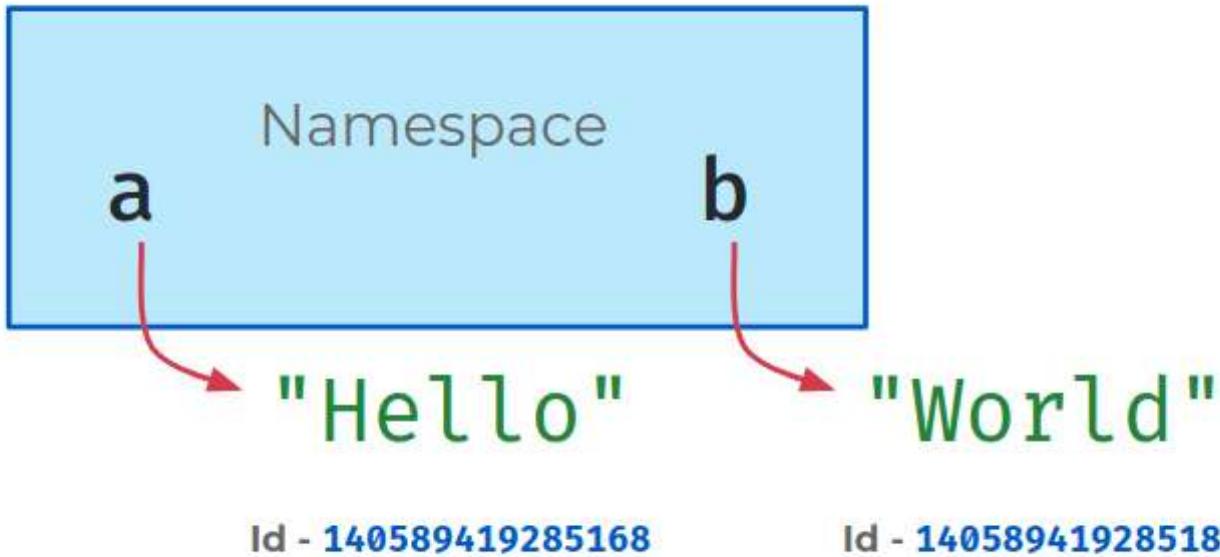
```
a = "Hello"  
b = "World"
```



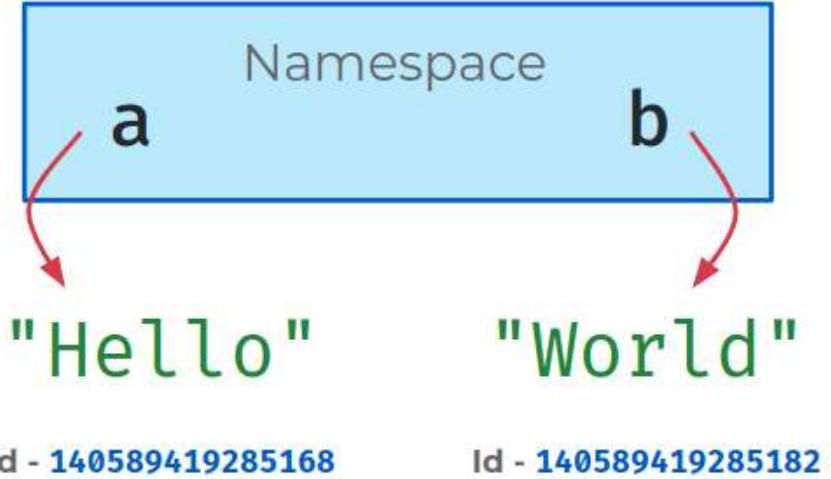
### Namespaces

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

PYTHON

```

1 def greet_1():
2     a = "Hello"
3     print(a)
4     print(id(a))
5
6 def greet_2():
7     a = "Hey"
8     print(a)
9     print(id(a))
10
11 print("Namespace - 1")
12 greet_1()
13 print("Namespace - 2")
14 greet_2()

```

Collapse ▲

## 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.

## Code

```
import math  
a = 2  
  
def foo():  
    b = 3  
    print(a + b)  
  
foo()
```



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.

## Code

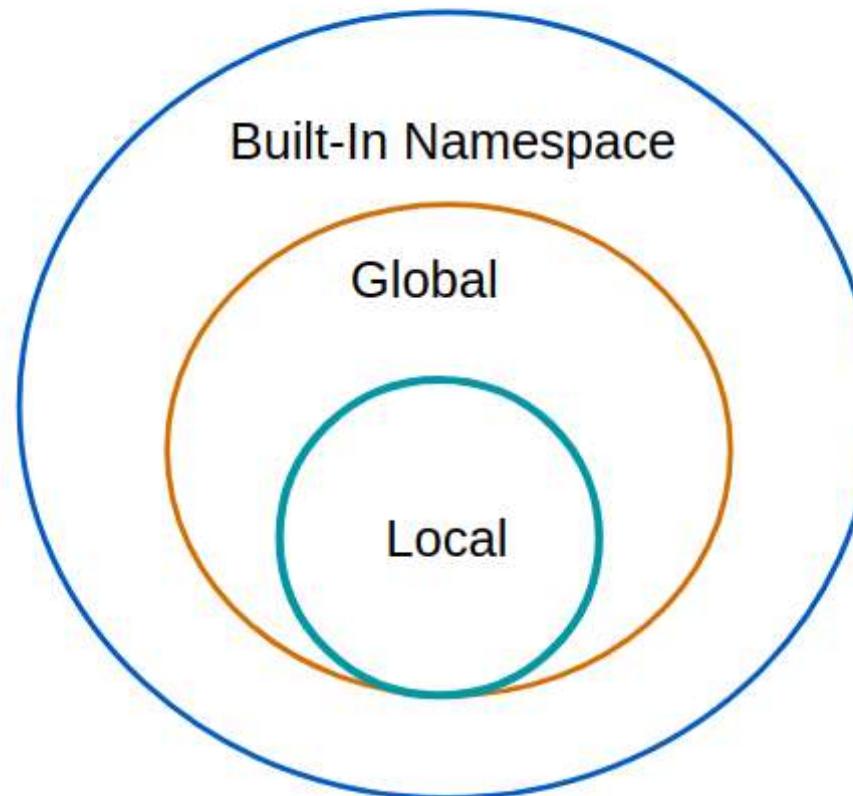
```
import math  
a = 2  
  
def foo():  
    b = 3  
    print(a + b)  
  
foo()
```



## 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

PYTHON

```
1 x = "Global Variable"
2 print(x)
3
4 def foo():
5     print(x)
6
7 foo()
```

Output

```
Global Variable
Global Variable
```

### *Example 2*

Code

PYTHON

```
1 def foo():
2     print(x)
3
4 x = "Global Variable"
5
6 foo()
```

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

```
def foo():
    x = "Local Variable"
    print(x)
```

```
foo()
print(x)
```

Scope of x



### Code

```
1 def foo():
2     x = "Local Variable"
```

PYTHON

```
3     print(x)
4
5 foo()
6 print(x)
```

## Output

```
Local Variable
NameError: name 'x' is not defined
```

As,

x is not declared before assignment, python throws an error.

## Local Import

### Code

PYTHON

```
1 def foo():
2     import math
3     print(math.pi)
4
5 foo()
6 print(math.pi)
```

## Output

```
3.141592653589793
NameError: name 'math' is not defined
```

## Local Variables & Global Variables

## Code

PYTHON

```
1 x = "Global Variable"
2
3 def foo():
4     x = "Local Variable"
5     print(x)
6
7 print(x)
8 foo()
9 print(x)
```

## 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

PYTHON

```
1 x = "Global Variable"
2
3 def foo():
4     global x
5     x = "Global Change"
6     print(x)
7
8 print(x)
9 foo()
```

```
10 print(x)
```

### Output

```
Global Variable  
Global Change  
Global Change
```

 MARKED AS COMPLETE

[Submit Feedback](#)

## 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 print("Hello")
2
3 def greet():
4     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 def divide(a, b):  
2     return a / b  
3
```

PYTHON

```
4 divide(5, 0)
```

## Output

```
ZeroDivisionError: division by zero
```

### Example 2

Input given by the user is not within expected values.

## Code

PYTHON

```
1
2 def divide(a, b):
3     return a / b
4
5 divide("5", "10")
```

## 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

PYTHON

```
1 class Store:
2     def __init__(self):
3         self.items = {
```

```
4         "milk" : 20, "bread" : 30, }
5
6     def add_item(self, name, quantity):
7         self.items[name] += quantity
8
9 s = Store()
10 s.add_item('biscuits', 10)
```

## 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.



Develop a Class

to model Bank Account

Use Bank Account class

to implement Money Transfer

Bank Account Class

*Example 1*

User 1

Balance

250/-

User 2

Balance

100/-

```
transfer_amount(user_1, user_2, 50)
```

User 1

Balance

200/-

User 2

Balance

150/-

Code

PYTHON

```
1 v class BankAccount:  
2 v     def __init__(self, account_number):  
3 v         self.account_number = str(account_number)  
4 v         self.balance = 0  
5 v     def get_balance(self):  
6 v         return self.balance  
7 v  
8 v
```



```
8
9     def withdraw(self, amount):
10    if self.balance >= amount:
11        self.balance -= amount
12    else:
13        print("Insufficient Funds")
14
15    def deposit(self, amount):
16        self.balance += amount
17
18
19 def transfer_amount(acc_1, acc_2, amount):
20     acc_1.withdraw(amount)
21     acc_2.deposit(amount)
22
23
24 user_1 = BankAccount("001")
25 user_2 = BankAccount("002")
26 user_1.deposit(250)
27 user_2.deposit(100)
28
29 print("User 1 Balance: {}/-".format(user_1.get_balance()))
30 print("User 2 Balance: {}/-".format(user_2.get_balance()))
31 transfer_amount(user_1, user_2, 50)
32 print("Transferring 50/- from User 1 to User 2")
33 print("User 1 Balance: {}/-".format(user_1.get_balance()))
34 print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ^

## 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

User 1

Balance

25/-

User 2

Balance

100/-

```
transfer_amount(user_1, user_2, 50)
```

User 1

Balance

25/-

User 2

Balance

150/-



Powered by  
**NXT WAVE** | IB HUBS

Code

PYTHON

```
1 -> class BankAccount:  
2 ->     def __init__(self, account_number):  
3 ->         self.account_number = str(account_number)  
4 ->         self.balance = 0
```

```
5
6     def get_balance(self):
7         return self.balance
8
9     def withdraw(self, amount):
10    if self.balance >= amount:
11        self.balance -= amount
12    else:
13        print("Insufficient Funds")
14
15    def deposit(self, amount):
16        self.balance += amount
17
18
19 def transfer_amount(acc_1, acc_2, amount):
20     acc_1.withdraw(amount)
21     acc_2.deposit(amount)
22
23
24 user_1 = BankAccount("001")
25 user_2 = BankAccount("002")
26 user_1.deposit(25)
27 user_2.deposit(100)
28
29 print("User 1 Balance: {}/-".format(user_1.get_balance()))
30 print("User 2 Balance: {}/-".format(user_2.get_balance()))
31 transfer_amount(user_1, user_2, 50)
32 print("Transferring 50/- from User 1 to User 2")
33 print("User 1 Balance: {}/-".format(user_1.get_balance()))
34 print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ▲

## 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/-

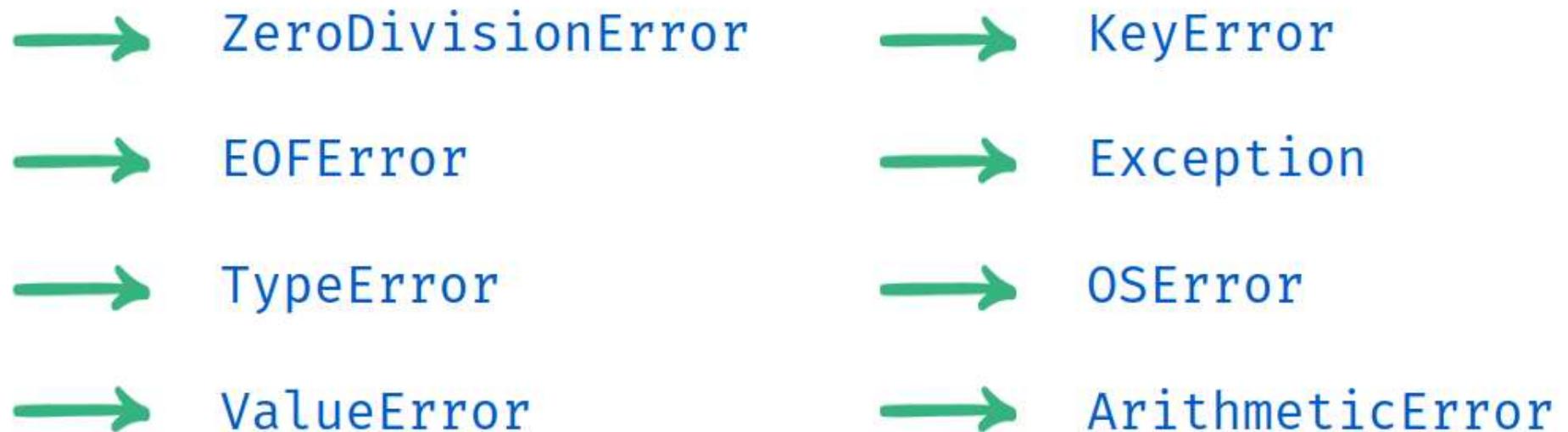
## Raising Exceptions

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.



and many more ...



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

User 1

Balance

25/-

User 2

Balance

100/-

```
transfer_amount(user_1, user_2, 50)
```

**ValueError:** Insufficient Funds



Code

PYTHON

```
1 class BankAccount:  
2     def __init__(self, account_number):  
3         self.account_number = str(account_number)  
4         self.balance = 0  
5  
6     def get_balance(self):  
7         return self.balance  
8  
9     def withdraw(self, amount):  
10        if self.balance >= amount:  
11            self.balance -= amount  
12        else:  
13            raise ValueError("Insufficient Funds")  
14  
15    def deposit(self, amount):  
16        self.balance += amount  
17  
18  
19    def transfer_amount(acc_1, acc_2, amount):  
20        acc_1.withdraw(amount)  
21        acc_2.deposit(amount)  
22  
23  
24 user_1 = BankAccount("001")  
25 user_2 = BankAccount("002")  
26 user_1.deposit(25)  
27 user_2.deposit(100)  
28  
29 print("User 1 Balance: {}/-".format(user_1.get_balance()))  
30 print("User 2 Balance: {}/-".format(user_2.get_balance()))  
31 transfer_amount(user_1, user_2, 50)  
32 print("Transferring 50/- from User 1 to User 2")  
33 print("User 1 Balance: {}/-".format(user_1.get_balance()))  
34 print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ^

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.

PYTHON

### Transfer Amount

### *Example 1*

User 1

Balance

25/-

User 2

Balance

100/-

transfer\_amount(user\_1, user\_2, 50)



User 1

Balance

25/-

User 2

Balance

100/-



Code

PYTHON

```
1 -> class BankAccount:  
2 ->     def __init__(self, account_number):  
3 ->         self.account_number = str(account_number)  
4 ->         self.balance = 0  
5 ->  
6 ->     def get_balance(self):  
7 ->         return self.balance  
8 ->  
9 ->     def withdraw(self, amount):  
10 ->         if amount <= self.balance:  
11 ->             self.balance -= amount  
12 ->         else:  
13 ->             print("Insufficient balance")
```

```
10     if self.balance >= amount:
11         self.balance -= amount
12     else:
13         raise ValueError("Insufficient Funds")
14
15     def deposit(self, amount):
16         self.balance += amount
17
18
19 def transfer_amount(acc_1, acc_2, amount):
20     try:
21         acc_1.withdraw(amount)
22         acc_2.deposit(amount)
23         return True
24     except:
25         return False
26
27
28 user_1 = BankAccount("001")
29 user_2 = BankAccount("002")
30 user_1.deposit(25)
31 user_2.deposit(100)
32
33 print("User 1 Balance: {}/-".format(user_1.get_balance()))
34 print("User 2 Balance: {}/-".format(user_2.get_balance()))
35 print(transfer_amount(user_1, user_2, 50))
36 print("Transferring 50/- from User 1 to User 2")
37 print("User 1 Balance: {}/-".format(user_1.get_balance()))
38 print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ▲

## 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

PYTHON

```
1 try:  
2     # Write code that  
3     # might cause exceptions.  
4 except Exception:  
5     # The code to be run when  
6     # there is an exception.
```

#### Example 1

#### Code

PYTHON

```
1 try:  
2     a = int(input())  
3     b = int(input())  
4     c = a/b  
5     print(c)  
6 except ZeroDivisionError:
```

```
7     print("Denominator can't be 0")
8 except:
9     print("Unhandled Exception")
```

### Input

```
5
0
```

### Output

```
Denominator can't be 0
```

### Example 2

#### Code

Input given by the user is not within expected values.

PYTHON

```
1 try:
2     a = int(input())
3     b = int(input())
4     c = a/b
5     print(c)
6 except ZeroDivisionError:
7     print("Denominator can't be 0")
8 except:
9     print("Unhandled Exception")
```

### Input

12

a

## Output

Unhandled Exception

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

## Syntax

PYTHON

```
1 try:
2     # Write code that
3     # might cause exceptions.
4 except Exception as e:
5     # The code to be run when
6     # there is an exception.
```

## Code

PYTHON

```
1 class BankAccount:
2     def __init__(self, account_number):
3         self.account_number = str(account_number)
4         self.balance = 0
5
6     def get_balance(self):
7         return self.balance
8
9     def withdraw(self, amount):
10        if self.balance >= amount:
11            self.balance -= amount
12        else:
```

```
13         raise ValueError("Insufficient Funds")
14
15     def deposit(self, amount):
16         self.balance += amount
17
18
19     def transfer_amount(acc_1, acc_2, amount):
20         try:
21             acc_1.withdraw(amount)
22             acc_2.deposit(amount)
23             return True
24         except ValueError as e:
25             print(str(e))
26             print(type(e))
27             print(e.args)
28             return False
29
30 user_1 = BankAccount("001")
31 user_2 = BankAccount("002")
32 user_1.deposit(25)
33 user_2.deposit(100)
34
35 print("User 1 Balance: {}/-".format(user_1.get_balance()))
36 print("User 2 Balance: {}/-".format(user_2.get_balance()))
37 print(transfer_amount(user_1, user_2, 50))
38 print("Transferring 50/- from User 1 to User 2")
39 print("User 1 Balance: {}/-".format(user_1.get_balance()))
40 print("User 2 Balance: {}/-".format(user_2.get_balance()))
```

Collapse ▲

## Output

```
User 1 Balance: 25/-
User 2 Balance: 100/-
Insufficient Funds
<class 'ValueError'>
('Insufficient Funds')
```

```
\ insufficient funds )\nFalse\nTransferring 50/- from User 1 to User 2\nUser 1 Balance: 25/-\nUser 2 Balance: 100/-
```

## Handling Multiple Exceptions

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

### Syntax

PYTHON

```
1 try:\n2     # Write code that\n3     # might cause exceptions.\n4 except Exception1:\n5     # The code to be run when\n6     # there is an exception.\n7 except Exception2:\n8     # The code to be run when\n9     # there is an exception.
```

### Example 1

### Code

PYTHON

```
1 try:\n2     a = int(input())\n3     b = int(input())\n4     c = a/b\n5     print(c)\n6 except ZeroDivisionError:\n7     print("Denominator can't be 0")\n8 except ValueError:\n9     print("Input should be an integer")
```

```
10    except:  
11        print("Something went wrong")
```

Collapse ▲

### Input

```
5  
0
```

### Output

```
Denominator can't be 0
```

### Example 2

#### Code

PYTHON

```
1    try:  
2        a = int(input())  
3        b = int(input())  
4        c = a/b  
5        print(c)  
6    except ZeroDivisionError:  
7        print("Denominator can't be 0")  
8    except ValueError:  
9        print("Input should be an integer")  
10   except:  
11        print("Something went wrong")
```

Collapse ▲

### Input

```
12
```

a

Output

Input should be an integer



MARKED AS COMPLETE

[Submit Feedback](#)

## 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 import datetime
2
3 date_object = datetime.date(2019, 4, 13)
4 print(date_object)
```

PYTHON

## Output

```
2019-04-13
```

## Date Object

### Code

PYTHON

```
1 from datetime import date  
2 date_obj = date(2022, 2, 31)  
3 print(date_obj)
```

## Output

```
ValueError: day is out of range for month
```

## Today's Date

### Class method

`today()` returns a date object with **today's date**.

### Code

PYTHON

```
1 import datetime  
2  
3 date_object = datetime.date.today()  
4 print(date_object)
```

## Output

```
2021-02-05
```

## Attributes of Date Object

### Code

PYTHON

```
1 from datetime import date  
2  
3 date_object = date(2019, 4, 13)  
4 print(date_object.year)  
5 print(date_object.month)  
6 print(date_object.day)
```

## 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

PYTHON

```
1 from datetime import time  
2  
3 time_object = time(11, 34, 56)
```

```
4 print(time_object)
```

## Output

```
11:34:56
```

## Attributes of Time Object

### Code

PYTHON

```
1 from datetime import time  
2  
3 time_object = time(11, 34, 56)  
4 print(time_object)  
5 print(time_object.hour)  
6 print(time_object.minute)  
7 print(time_object.second)
```

## 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

PYTHON

```
1 from datetime import datetime  
2  
3 date_time_obj = datetime(2018, 11, 28, 10, 15, 26)  
4 print(date_time_obj.year)  
5 print(date_time_obj.month)  
6 print(date_time_obj.hour)  
7 print(date_time_obj.minute)
```

Output

```
2018  
11  
10  
15
```

*Example - 2*

It gives the current date and time

Code

PYTHON

```
1 import datetime  
2  
3 datetime_object = datetime.datetime.now()  
4 print(datetime_object)
```

Output

2021-02-05 09:26:08.077473

## DateTime object

### Code

PYTHON

```
1 from datetime import datetime  
2 date_time_obj = datetime(2018, 11, 28)  
3 print(date_time_obj)
```

### 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, ...

Format Specifier	Meaning	Example
%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

PYTHON

```

1 from datetime import datetime
2
3 now = datetime.now()
4 formatted_datetime_1 = now.strftime("%d %b %Y %I:%M:%S %p")
5 print(formatted_datetime_1)
6
7 formatted_datetime_2 = now.strftime("%d/%m/%Y, %H:%M:%S")
8 print(formatted_datetime_2)

```

## 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

PYTHON

```
1 from datetime import datetime  
2  
3 date_string = "28 November, 2018"  
4 print(date_string)  
5  
6 date_object = datetime.strptime(date_string, "%d %B, %Y")  
7 print(date_object)
```

Output

```
28 November, 2018  
2018-11-28 00:00:00
```

## Working with 'timedelta' Class

Timedelta object represents **duration**.

*Example 1*

Code

PYTHON

```
1 from datetime import timedelta
```

```
2
3 delta = timedelta(days=365, hours=4)
4 print(delta)
```

## Output

```
365 days, 4:00:00
```

## Example 2

### Code

PYTHON

```
1 from datetime import timedelta, datetime
2 delta = timedelta(days=365)
3 current_datetime = datetime.now()
4 print(current_datetime)
5 next_year_datetime = current_datetime + delta
6 print(next_year_datetime)
```

## Output

```
2021-02-05 09:28:30.239095
2022-02-05 09:28:30.239095
```

## Calculating Time Difference

### Code

PYTHON

```
1 import datetime
2
3 dt1 = datetime.datetime(2021, 2, 5)
```

```
4 dt2 = datetime.datetime(2022, 1, 1)
5 duration = dt2 - dt1
6 print(duration)
7 print(type(duration))
```

## Output

```
330 days, 0:00:00
<class 'datetime.timedelta'>
```

 MARKED AS COMPLETE

[Submit Feedback](#)