



MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Dr. Vishwanath Karad

**MIT WORLD PEACE
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Unit-IV

4 Major Versions of Python

- “Python” or “CPython” is written in C/C++
 - Version 2.7 came out in mid-2010
 - Version 3.1.2 came out in early 2010
- “Jython” is written in Java for the JVM
- “IronPython” is written in C# for the .Net environment

Contd...

- Created in 1989 by Guido Van Rossum
- Python 1.0 released in 1994
- Python 2.0 released in 2000
- Python 3.0 released in 2008
- Python 2.7 is the recommended version
- 3.0 adoption will take a few years



Development Environments IDE

1. PyDev with Eclipse
2. Komodo
3. Emacs
4. Vim
5. TextMate
6. Gedit
7. Idle
8. PIDA (Linux)(VIM Based)
9. NotePad++ (Windows)
10. Pycharm

Web Frameworks

- Django
- Flask
- Pylons
- TurboGears
- Zope
- Grok

Introduction

- Multi-purpose (Web, GUI, Scripting, etc.)
- Object Oriented
- Interpreted
- Strongly typed and Dynamically typed
- Focus on readability and productivity

Python features

- no compiling or linking
- rapid development cycle
- no type declarations
- simpler, shorter, more flexible
- automatic memory management
- garbage collection
- high-level data types and operations

Contd.

.

- fast development
- object-oriented programming
- code structuring and reuse, C++
- embedding and extending in C
- mixed language systems
- classes, modules, exceptions, multithreading
- "programming-in-the-large" support

Uses of Python

- shell tools
 - system admin tools, command line programs
- extension-language work
- rapid prototyping and development
- language-based modules
 - instead of special-purpose parsers
- graphical user interfaces
- database access
- distributed programming
- Internet scripting



Why Python

- Python is a general purpose, high level object oriented language.
- It is an interpreted language.
- It is an open source and free language.
- It is a dynamic programming language.




Before writing a program.....

- **\$ where is python**

python: /usr/bin/python /usr/bin/python2.7 /etc/python
/etc/python2.7 /usr/lib/python2.6 /usr/lib/python2.7
/usr/local/lib/python2.7 /usr/include/python2.6
/usr/include/python2.7 /usr/share/python
/usr/share/man/man1/python.1.gz

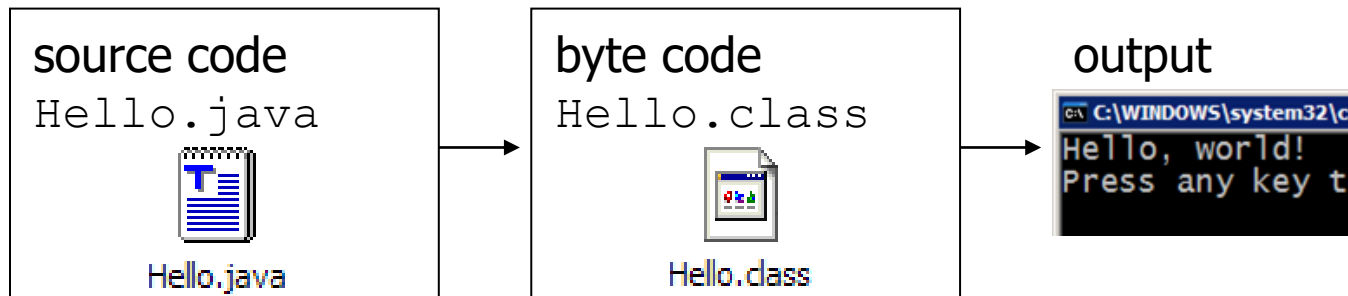


- 
- C:\WINDOWS\system32\cmd.exe
- Hello, world!
- Press any key to continue

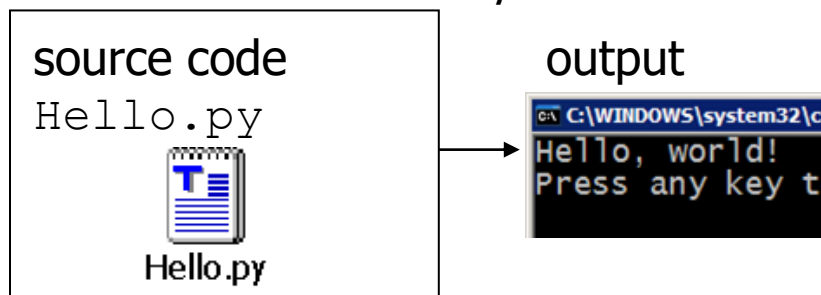


Compiling and Interpreting

Many languages require to *compile* (translate) program into a form that the machine understands *execute*



Python is instead directly *interpreted* into machine instructions.



Python Editors

- Onlinegdb(Online)
- Jupyter(Online)
- Anaconda
- Python IDE

First Python Program

- `print ("Hello, World!")`

```
comp197@comp197:~/Desktop$ python  
ass.py
```

```
Hello world
```

Using variables

```
i = 5
```

```
print(i)
```

```
i = i + 1
```

```
print(i)
```

```
s = """This is a multi-line string.
```

```
This is the second line."""
```

```
print(s)
```


Output

```
$ python var.py
```

5

6

This is a multi-line string.

This is the second line.

Comments in Python

```
#!/usr/bin/python
```

```
# First comment
```

```
print "Hello, Python!"; # second comment
```

Indentation-Most Important

```
i = 5
```

```
    print('Value is ', i) # Error! Notice a single space  
at the start of the line
```

```
print('I repeat, the value is ', i)
```

Using Expressions

```
#!/usr/bin/python
```

```
# Filename: expression.py
```

```
length = 5
```

```
breadth = 2
```

```
area = length * breadth
```

```
print('Area is', area)
```

```
print('Perimeter is', 2 * (length + breadth))
```

Output

```
$ python expression.py
```

```
Area is 10
```

```
Perimeter is 14
```

Math commands

Python has useful commands for performing calculations.

To use math commands, you must write the following at the top of your Python program:

```
from math import *
```

Command name	Description
<code>abs(value)</code>	absolute value
<code>ceil(value)</code>	rounds up
<code>cos(value)</code>	cosine, in radians
<code>floor(value)</code>	rounds down
<code>log(value)</code>	logarithm, base e
<code>log10(value)</code>	logarithm, base 10
<code>max(value1, value2)</code>	larger of two values
<code>min(value1, value2)</code>	smaller of two values
<code>round(value)</code>	nearest whole number
<code>sin(value)</code>	sine, in radians
<code>sqrt(value)</code>	square root

Variables

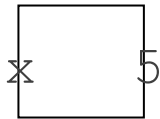
variable: A named piece of memory that can store a value.

- Compute an expression's result,
- store that result into a variable,
- and use that variable later in the program.

assignment statement: Stores a value into a variable.

- Syntax: *name* = *value*

- Examples: $x = 5$ $gpa = 3.14$



- A variable that has been given a value can be used in expressions.

$x + 4$ is 9

Exercise: Evaluate the multiplication of three numbers for a given a , b , and c .

Variables

- Can contain letters, numbers, and underscores
- Must begin with a letter
- Cannot be one of the reserved Python keywords:
and, as, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if, import,
in, is, lambda, not, or, pass, print, raise, return, try,
while, with, yield

Variable types

- Example types.py:

```
pi = 3.1415926
```

```
message = "Hello,  
world"
```

```
i = 2+2
```

```
print type(pi)
```

```
print type(message)
```

```
print type(i)
```

- Output:

```
<type 'float'>
```

```
<type 'str'>
```

```
<type 'int'>
```

- Python has incorporated operators like +=,

but ++ (or --) do not work in Python

Variable types

- `int()`, `float()`, `str()`, and `bool()` convert to integer, floating point, string, and boolean (True or False) types, respectively

- Example `typeconv.py`:

```
print 1.0/2.0  
print 1/2  
print float(1)/float(2)  
print int(3.1415926)  
print str(3.1415926)  
print bool(1)  
print bool(0)
```

- Output:

```
0.5  
0  
0.5  
3  
3.141592  
6  
True  
False
```



print (display on console)

print : Produces text output on the console.

Syntax:

```
print "Message"
```

```
print Expression
```

- Prints the given text message or expression value on the console, and moves the cursor down to the next line.

```
print Item1, Item2, ..., ItemN
```

- Prints several messages and/or expressions on the same line.

Examples:

```
print "Hello, world!"
```

```
age = 45
```

```
print "You have", 65 - age, "years  
until retirement"
```

Output:

```
Hello, world!
```

```
You have 20 years until retirement
```

input (read)

input : Reads a number from user input.

- You can assign (store) the result of **input** into a variable.

- Example:

```
age = input("How old are you? ")  
i = raw_input("Enter a math expression: ")  
print "Your age is", age  
print "You have", 65 - age, "years until  
retirement"
```

Output:

```
How old are you? 53  
Enter a math expression: 2+5  
Your age is 53  
You have 12 years until retirement
```

Exercise: Write a Python program to accept the marks of three subjects from user and display its average.

Basic operations

- Assignment:

```
- size =  
40 a = b = c = 3
```

- Numbers

- integer, float
- complex numbers: `1j+3`, `abs(z)`

- Strings

- `'hello world'`, `'it\'s hot'`
- `"bye world"`

String operations

- concatenate with + or neighbours

– `word = 'Help' + x`

– `word = 'Help' 'a'`

- subscripting of strings

– `'Hello'[2] □ 'l'`

– slice: `'Hello'[1:2] □ 'el'`

– `word[-1] □ last character`

– `len(word) □ 5`

– immutable: cannot assign to subscript

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```



Numbers: Floating Point

- `int(x)` converts `x` to an integer
- `float(x)` converts `x` to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.232320000000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```


Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are *immutable*

```
>>> x = 4.5
```

```
>>> y = x
```

```
>>> y += 3
```

```
>>> x
```

```
4.5
```

```
>>> y
```

```
7.5
```

x → 4.5

y ↗

x → 4.5

y → 7.5

String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'  
>>> x = x + ' there'  
>>> x  
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)'
>>> print s
And me too!
though I am much longer
than the others :)'
```

Substrings and Methods

```
>>> s = '012345'  
>>> s[3]  
'3'  
>>> s[1:4]  
'123'  
>>> s[2:]  
'2345'  
>>> s[:4]  
'0123'  
>>> s[-2]  
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)  
6  
>>> str(10.3)  
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
```

```
    pass
```

Operators

- Arithmetic

a	=	10	#	10
a	+=	1	#	11
a	-=	1	#	10
b	=	a + 1	#	11
c	=	a - 1	#	9
d	=	a * 2	#	20
e	=	a / 2	#	5
f	=	a % 3	#	1
g	=	a ** 2	#	100

String Manipulation

```
animals = "Cats " + "Dogs "  
animals += "Rabbits"  
# Cats Dogs Rabbits  
  
fruit = ', '.join(['Apple', 'Banana', 'Orange'])  
# Apple, Banana, Orange  
  
date = '%s %d %d' % ('Sept', 11, 2010)  
# Sept 11 2010  
  
name = '%(first)s %(last)s' % {  
    'first': 'Nowell',  
    'last': 'Strite'}  
# Nowell Strite
```

Logical Comparison

```
# Logical And  
a and b
```

```
# Logical Or  
a or b
```

```
# Logical Negation  
not a
```

```
# Compound  
(a and not (b or c))
```

Identity Comparison

```
# Identity
1 is 1 == True

# Non Identity
1 is not '1' == True

# Example
bool(1) == True
bool(True) == True

1 and True == True
1 is True == False
```

Arithmetic Comparison

Ordering

$a > b$

$a \geq b$

$a < b$

$a \leq b$

Equality/Difference

$a == b$

$a != b$

What is a List in Python?

- A **list** is a data structure that's built into Python and holds a collection of items. Lists have a number of important characteristics:
- List items are enclosed in square brackets, like this *[item1, item2, item3]*.
- Lists are **ordered** – i.e. the items in the list appear in a specific order. This enables us to use an index to access to any item.
- Lists are **mutable**, which means you can add or remove items after a list's creation.
- List elements **do not need to be unique**. Item duplication is possible, as each element has its own distinct place and can be accessed separately through the index.
- Elements can be of **different data types**: you can combine strings, integers, and objects in the same list.

Lists in Python

- If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:
- `car1 = "Ford"`
`car2 = "Volvo"`
`car3 = "BMW"`
- `cars = ["Ford", "Volvo", "BMW"]`

Printing list

- `print(cars)`
- Or
- `for x in cars:`
 `print(x)`
- `for i in range(1,4):`
- `print(cars[i])`

Built in Functions of list

- `a=[1,2,3,4,5]`
- `d=[2,2.3,4.5,6.7]`
- `Print(max(a))`
- `Print(min(a))`
- `Print(sorted(a))`
- `len(a)`
- `a.pop()`
- `a.remove(3)`
- `del a[2]`

Accepting list from user

- `n=int(input("enter size"))`
- `a=[1,2,3,4,5]`
- `d=[2,2.3,4.5,6.7]`
- `for i in range(0,n):`
- `x=int(input("enter a number"))`
- `#a.append(x)`
- `a[i]=x`

Problems on list

- Calculate sum of n elements
- Adding element in list at particular position
- Printing list in reverse order

Remove duplicate elements from list

Python Data Types: Dictionary

```
course
={'PPL':'Jayshree','DS':'Priyanka','COA':'Shamla'}
print("dictionary :", course)
print(course.values())
print(course.keys())
print(course['DS']) # access an element
course["OOP"]="Shreya"
print("dictionary :", course)
print("length dictionary :",len(course))
course["OOP"]="Object Prog" # change value
print("dictionary :", course)
```

List Operations

There are various operations that we can perform on Lists.

1. Update elements

There are several ways you can add elements to a list.

```
orderItem=[1, "Sam", "Computer", 75.50, True]
```

```
Print(orderItem)
```

```
orderItem[2]="Laptop" #addition of
```

```
Print(orderItem)
```

2. Adding item at the desired location

```
orderItem=[1, "Sam", "Computer", 75.50, True]
```

```
OrderItem.insert(3, Personal) # adding element
```

```
100 at the fourth location
```

3. #Adding element at the end of the list

```
orderItem=[1, "Sam", "Computer", 75.50, True]
orderItem.append('abc')
```

4. #Adding several elements at the end of list

```
orderItem=[1, "Sam", "Computer", 75.50, True]
orderItem.extend(['MIT',2020])
```

Delete elements

Deleting 2nd element

```
orderItem=[1, "Sam", "Computer", 75.50, True]
del orderItem[1]
```

Deleting elements from 3rd to 4th

```
orderItem=[1, "Sam", "Computer", 75.50, True]
del orderItem[2:3]
```

Deleting the whole list

#remove(item): Removes specified item from list.

```
orderItem=[1, "Sam", "Computer", 75.50, True]
orderItem.remove('C')
```

#pop(index): Removes the element from the given index.

```
# Deleting 2nd element
orderItem=[1, "Sam", "Computer", 75.50, True]
orderItem.pop(1)
```

#Deleting all the elements

```
orderItem=[1, "Sam", "Computer", 75.50, True]
orderItem.clear()
```

Find the sum of 'n' numbers

```
n = input("Enter the number of elements to be calculated for  
finding the sum :")
```

```
n = int (n)
```

```
sum = 0
```

```
for num in range(0,n+1,1):
```

```
    sum = sum + num
```

```
print("SUM of first ", n, "numbers is: ", sum )
```

range (start, stop, step)

It takes three arguments. Out of the three 2 arguments are optional. I.e., start and step are the optional arguments.

A start argument is a starting number of the sequence. i.e., **lower limit**. By default, it starts with 0 if not specified.

A stop argument is an **upper limit**. i.e., generate numbers up to this number, The range() doesn't include this number in the result.

The step is a **difference between each number in the result**. The default value of the step is 1 if not specified.

Python Arrays

Python does not have built-in support for Arrays, but Python Lists can be used instead.

Arrays are used to store multiple values in one single variable

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value

#Find the average of elements of an array

```
n = int(input("enter the size of array :"))
arr = [ ]
sum=0
for i in range(n):
    x = int(input("enter element: "))
    arr.append(x)
    sum=sum+arr[i]
print("Entered array is :", arr, " and Average of elements is ",
float(round((sum/n),2)))
```


Python Data Types: Tuples

- In Python, a tuple is similar to List except that the objects in tuple are immutable which means we cannot change the elements of a tuple once assigned. On the other hand, we can change the elements of a list.
- **Tuple vs List**
- 1. The elements of a **list** are mutable whereas the elements of a **tuple** are immutable.
- 2. When we do not want to change the data over time, the **tuple** is a preferred data type whereas when we need to change the data in future, **list** would be a wise option.
- 3. Iterating over the elements of a **tuple** is faster compared to

How to create a tuple in Python

- A tuple is another sequence data type that is similar to the list.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated.

- **TupSub = ('DMTA', 'NLP' , 'CS121')**
- **TupMob = ('Iphone6', 'Sony', 'Appo')**
- **print (TupSub) # Prints complete list**
- **print (TupSub[0]) # Prints first element of the list**
- **print (TupSub[1:3]) # Prints elements starting from 2nd till 3rd**
- **print (TupSub[2:]) # Prints elements starting from 3rd element**
- **print(TupMob * 2) # Prints list two times**

Accessing tuple elements using positive indexes

```
# tuple of strings  
my_data = ("hi", "hello",  
"bye")
```

```
# displaying all elements  
print(my_data)
```

```
# accessing first element  
# prints "hi"  
print(my_data[0])
```

```
# accessing third element  
# prints "bye"  
print(my_data[2])
```

Negative indexes in tuples

Similar to list and strings we can use negative indexes to access the tuple elements from the end.

-1 to access last element, -2 to access second last and so on.

```
my_data = (1, 2, "Kevin", 8.9)
```

```
# accessing last element  
# prints 8.9  
print(my_data[-1])
```

```
# prints 2  
print(my_data[-3])
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely:

Example

The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")  
del thistuple  
print(thistuple) #this will raise an error because the  
tuple no longer exists
```

Built-in Tuple Methods

len(): Returns the number of elements in the tuple.

```
t1=(12,45,43,8,35)
len(t1)
```

max(): If the tuple contains numbers, the heighest number will be returned. If the tuple contains strings, the one that comes last in alphabetical order will be returned.

```
t1=(12, 45, 43, 8, 35)
max(t1)
t2=('python', 'java', 'C++')
max(t2)
```

min(): If the tuple contains numbers, the lowest number will be returned. If the tuple contains strings, the one that comes first in alphabetical order will be returned.

```
t1=(12,45,43,8,35)
min(t1)
```

Built-In Methods

BUILT-IN FUNCTION	DESCRIPTION
all()	Returns true if all element are true or if tuple is empty
any()	return true if any element of the tuple is true. if tuple is empty, return false
len()	Returns length of the tuple or size of the tuple
enumerate()	Returns enumerate object of tuple
max()	return maximum element of given tuple
min()	return minimum element of given tuple
<u>sum()</u>	Sums up the numbers in the tuple
<u>sorted()</u>	input elements in the tuple and return a new sorted list
<u>tuple()</u>	Convert an iterable to a tuple.

Immutable and Mutable

- `TupSub = ('DMTA', 'NLP' , 'CS121')`
- `Code = [123, 124, 125]`
- `TupSub[2] = 'Compiler' # Invalid syntax with tuple`
- `Code[2] = 1000 # Valid syntax with list`

Python Data Types: Dictionary

- Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])
- ```
cardict = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
print(cardict)
```



## Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
x = cardict["model"]
```

There is also a method called **get()** that will give you the same result:

Example

Get the value of the "model" key:

```
x = cardict.get("model")
```

## Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
cardict = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
cardict["year"] = 2018
```

## Loop Through a Dictionary

You can loop through a dictionary by using a ***for loop***.

When looping through a dictionary, ***the return value are the keys of the dictionary***, but there are methods to return the values as well.

Example

Print all key names in the dictionary, one by one:

```
for x in cardict:
 print(x)
```

**Print all values in the dictionary,  
one by one:**

```
for x in cardict:
 print(cardict[x])
```

**You can also use the values()  
method to return values of a  
dictionary:**

```
for x in cardict.values():
 print(x)
```

**Loop through both keys and  
values, by using the items()  
method:**

```
for x, y in cardict.items():
 print(x, y)
```

## Python Dictionary clear() Method

### Example

Remove all elements from the car list:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
car.clear()
```

```
print(car)
```

## Python Dictionary copy() Method

### Example

Copy the car dictionary:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
x = car.copy()
```

```
print(x)
```

## Python Dictionary fromkeys() Method

### Example

Create a dictionary with 3 keys, all with the value 0:

```
x = ('key1', 'key2', 'key3')
```

```
y = 0
```

```
thisdict = dict.fromkeys(x, y)
```

```
print(thisdict)
```

## Python Dictionary items() Method

### Example

Return the dictionary's key-value pairs:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
x = car.items()
```

```
print(x)
```

## Python Dictionary keys() Method

Example

Return the keys:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
x = car.keys()
```

```
print(x)
```

## Python Dictionary pop() Method

Example

Remove "model" from the dictionary:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
car.pop("model")
```

```
print(car)
```

**car.popitem() #popitem()** Method

Remove the last item from the dictionary

```
print(car)
```



# Python Data Types: Dictionary

```
course = {'PPL':'Jayshree','DS':'Priyanka','COA':'Shamla'}
print("dictionary :", course)
print(course.values())
print(course.keys())
print(course['DS']) # acess an element
course["OOP"]="Shreya"
print("dictionary :", course)
print("length dictionary : ",len(course))
course["OOP"]="Object Prog" # change value
print("dictionary :", course)
del course["OOP"]
print("dictionary :", course)
```

# Sets

- A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.
- ```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```


Sets

- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- `thisset = {"apple", "banana", "cherry"}`

```
for x in thisset:  
    print(x)
```

Sets

- `thisset = {"apple", "banana", "cherry"}`
`print("banana" in thisset)`
- `thisset = {"apple", "banana", "cherry"}`
`thisset.add("orange")`
`print(thisset)`

- `thisset = {"apple", "banana", "cherry"}`

`thisset.update(["orange", "mango", "grapes"])`

`print(thisset)`

- `thisset = {"apple", "banana", "cherry"}`

`thisset.remove("banana")`

`print(thisset)`

- `thisset = {"apple", "banana", "cherry"}`

`thisset.discard("banana")`

`print(thisset)`

- `set1 = {"a", "b" , "c"}`
`set2 = {1, 2, 3}`

```
set3 = set1.union(set2)  
print(set3)
```

- `Intersection()`
- `Difference()`

- `set1 = {"a", "b" , "c"}`
`set2 = {1, 2, 3}`

```
set1.update(set2)  
print(set1)
```

- `set1 = set()`
- `print("Initial blank Set: ")`
- `print(set1)`
-
- `# Adding element and tuple to the Set`
- `set1.add(8)`
- `set1.add(9)`
- `set1.add((6,7))`
- `print("\nSet after Addition of Three elements: ")`
- `print(set1)`
-
- `# Adding elements to the Set`
- `# using Iterator`
- `for i in range(1, 6):`
- `set1.add(i)`
- `print("\nSet after Addition of elements from 1-5: ")`
- `print(set1)`

Python frozenset()

- The frozenset() function returns an immutable frozenset object initialized with elements from the given iterable.
- Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.
- Due to this, frozen sets can be used as keys in Dictionary or as elements of another set. But like sets, it is not ordered (the elements can be set at any index).
- The syntax of frozenset() function is:
- `frozenset([iterable])`

- `frozenset()` Parameters
- The `frozenset()` function takes a single parameter:
- `iterable` (Optional) - the iterable which contains elements to initialize the `frozenset` with.
- Iterable can be set, dictionary, tuple, etc.
- Return value from `frozenset()`
- The `frozenset()` function returns an immutable `frozenset` initialized with elements from the given iterable.
- If no parameters are passed, it returns an empty `frozenset`.

Example 1: Working of Python frozenset()

- # tuple of vowels
- vowels = ('a', 'e', 'i', 'o', 'u')
- fSet = frozenset(vowels)
- print('The frozen set is:', fSet)
- print('The empty frozen set is:',
frozenset())
- # frozensets are immutable
- fSet.add('v')

Output

The frozen set is: frozenset({'a', 'o', 'u', 'i', 'e'})

The empty frozen set is: frozenset()

Traceback (most recent call last):

File "<string>", line 8, in <module>
fSet.add('v')

AttributeError: 'frozenset' object has no attribute 'add'

Taking input in Python

- Often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.
- **input (prompt)**
- **raw_input (prompt)**
- **input ()** : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –

Control Structures: Conditional statement, Looping and Iteration,

- **Python if...else Statement**
- In this article, you will learn to create decisions in a Python program using different forms of if..else statement.
- Python if Statement Syntax
- if test expression:
- statement(s)

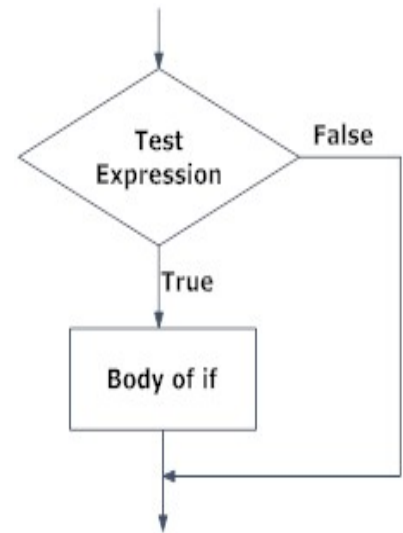


Fig: Operation of if statement

Example: Python if Statement

If the number is positive, we print an appropriate message

- `num = 3`
- `if num > 0:`
- `print(num, "is a positive number.")`
- `print("This is always printed.")`

Output

- `num = -1`
- `if num > 0:`
- `print(num, "is a positive number.")`
- `print("This is also always printed.")`
- When you run the program, the output will be:

3 is a positive number
This is always printed
This is also always printed.

- Python if...else Statement
- Syntax of if...else
- if test expression:
- Body of if
- else:
- Body of else

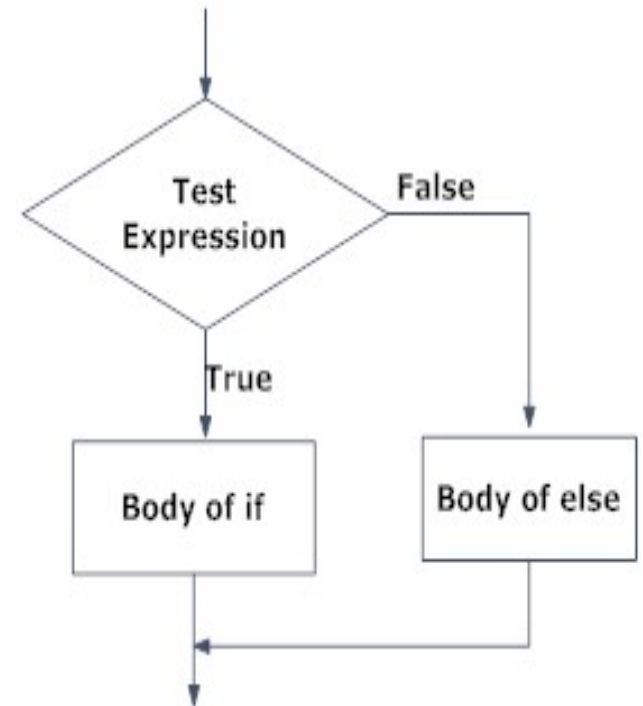


Fig: Operation of if...else statement

Example of if...else

Program checks if the number is positive or negative
And displays an appropriate message

- `num = 3`
- `# Try these two variations as well.`
- `# num = -5`
- `# num = 0`
- `if num >= 0:`
- `print("Positive or Zero")`
- `else:`
- `print("Negative number")`

Output

Positive or Zero

What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

- Syntax of for Loop
- for val in sequence:
- Body of for

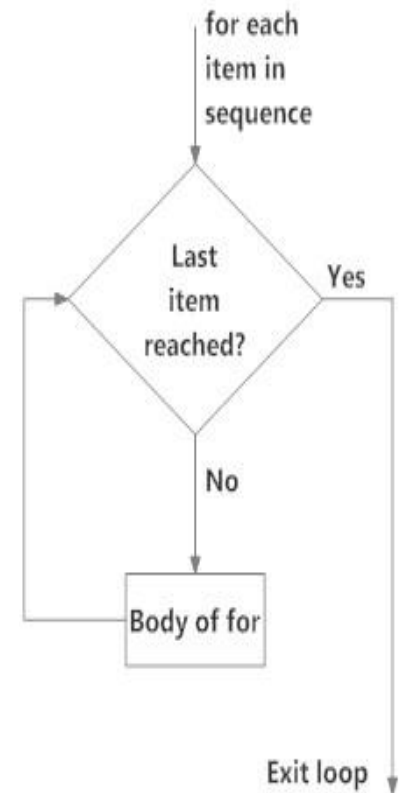


Fig: operation of for loop

Example: Python for Loop

Program to find the sum of all numbers
stored in a list

- # List of numbers
- numbers = [6, 5, 3, 8, 4, 2,
5, 4, 11]
- # variable to store the
sum
- sum = 0
- # iterate over the list
- for val in numbers:
- sum = sum+val
- print("The sum is", sum)

Output:

The sum is 48

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start, stop, step_size). step_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

```
print(range(10))
```

```
print(list(range(10)))
```

```
print(list(range(2, 8)))
```

```
print(list(range(2, 20, 3)))
```

Output

```
range(0, 10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[2, 3, 4, 5, 6, 7]
```

```
[2, 5, 8, 11, 14, 17]
```

What is while loop in Python?

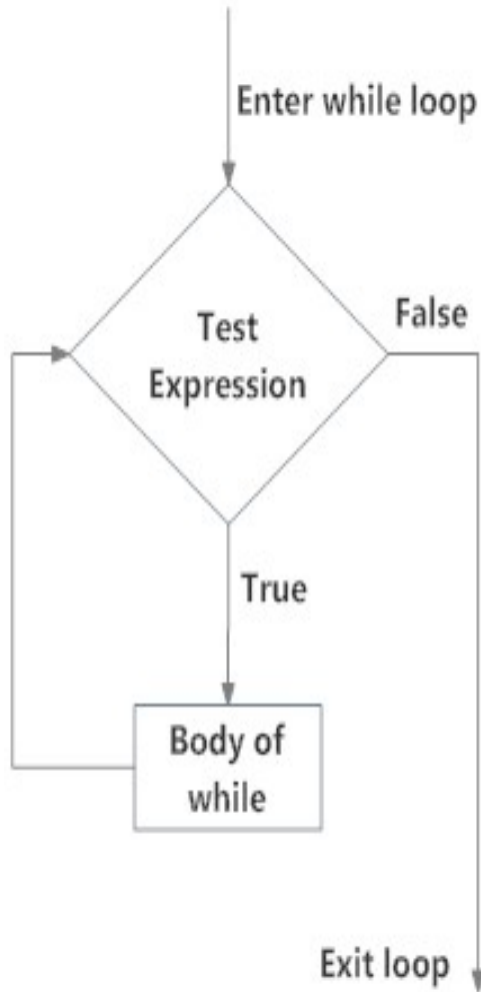


Fig: operation of while loop

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know the number of times to iterate beforehand.
- Syntax of while Loop in Python
- `while test_expression:`
- Body of while

Example: Python while Loop

Program to add natural

numbers up to # sum = 1+2+3+...+n

To take input from the user,

n = int(input("Enter n: "))

n = 10

initialize sum and counter

sum = 0

i = 1

while i <= n:

sum = sum + i

i = i+1 # update counter

print the sum

print("The sum is", sum)

When you run the program,
the output will be:

Enter n: 10

The sum is 55