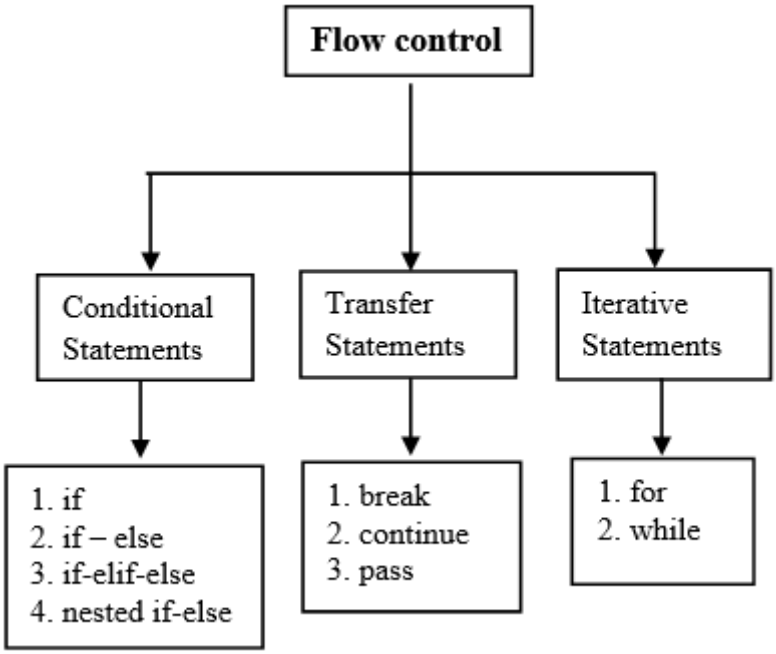
Python Control Flow Statements and Loops

In Python programming, flow control is the order in which statements or blocks of code are executed at runtime based on a condition.

The flow control statements are divided into **three categories**

1. Conditional statements
2. Iterative statements.
3. Transfer statements



Conditional statements

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.

There are three types of conditional statements.

1. if statement
2. if-else
3. if-elif-else
4. nested if-else

Iterative statements

In Python, iterative statements allow us to execute a block of code repeatedly as long as the condition is True. We also call it a loop statements.

Python provides us the following two loop statement to perform some actions repeatedly

1. for loop
2. while loop

Let's learn each one of them with the examples

Transfer statements

In Python, transfer statements are used to alter the program's way of execution in a certain manner. For this purpose, we use three types of transfer statements.

1. break statement
2. continue statement
3. `pass` statements

If statement in Python

In control statements, The `if` statement is the simplest form. It takes a condition and evaluates to either `True` or `False`.

If the condition is `True`, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to the next line

Syntax of the `if` statement

```
if condition:
     statement 1
     statement 2
     statement n
```

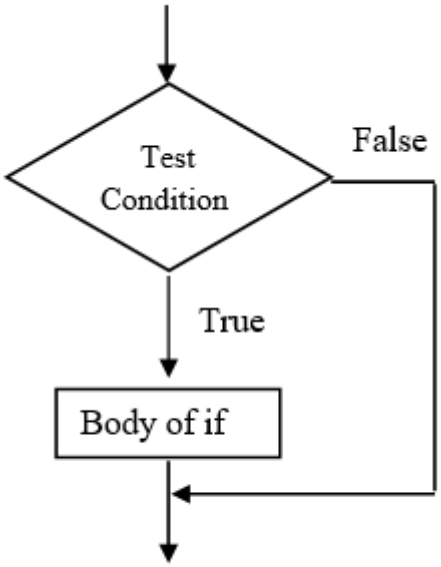


Fig. Flowchart of if statement

Example

```
number = 6
if number > 5:
    # Calculate square
    print(number * number)
print('Next lines of code')
```

Output

```
36
Next lines of code
```

If – else statement

The `if-else` statement checks the condition and executes the `if` block of code when the condition is True, and if the condition is False, it will execute the `else` block of code.

Syntax of the `if-else` statement

```
if condition:
    statement 1
else:
    statement 2
```

If the condition is `True`, then statement 1 will be executed If the condition is `False`, statement 2 will be executed. See the following flowchart for more detail.

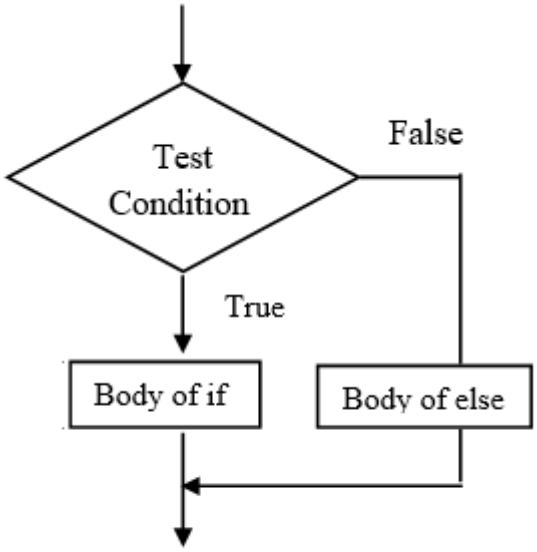


Fig. Flowchart of if-else

Example

```
password = input('Enter password ')

if password == "PYnative@#29":
    print("Correct password")
else:
    print("Incorrect Password")
```

Output 1

```
Enter password PYnative@#29
Correct password
```

Output 2

```
Enter password PYnative
Incorrect Password
```

Chain multiple if statement in Python

In Python, the `if-elif-else` condition statement has an `elif` blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.

With the help of `if-elif-else` we can make a tricky decision. The `elif` statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the `if-elif-else` statement:

```
if condition-1:
     statement 1
elif condition-2:
     stetement 2
elif condition-3:
     stetement 3
     ...
else:
     statement
```

Example

```
def user_check(choice):
    if choice == 1:
        print("Admin")
    elif choice == 2:
        print("Editor")
    elif choice == 3:
        print("Guest")
    else:
        print("Wrong entry")

user_check(1)
user_check(2)
user_check(3)
user_check(4)
```

Output:

```
Admin
Editor
Guest
Wrong entry
```

Nested if-else statement

In Python, the nested `if-else` statement is an if statement inside another `if-else` statement. It is allowed in Python to put any number of `if` statements in another `if` statement.

Indentation is the only way to differentiate the level of nesting. The nested `if-else` is useful when we want to make a series of decisions.

Syntax of the nested-`if-else`:

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers

```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))

if num1 >= num2:
    if num1 == num2:
        print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
else:
    print(num1, 'is smaller than', num2)
```

Output 1

```
Enter first number 56
Enter second number 15
56 is greater than 15
```

Output 2

```
Enter first number 29
Enter second number 78
29 is smaller than 78
```

For Loop

Syntax of `for` loop:

```
for element in sequence:
     body of for loop
```

Example to display first ten numbers using for loop

```
for i in range(1, 11):
    print(i)
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

While loop in Python

In Python, The while loop statement repeatedly executes a code block while a particular condition is true.

In a while-loop, every time the condition is checked at the beginning of the loop, and if it is true, then the loop's body gets executed. When the condition became False, the controller comes out of the block.

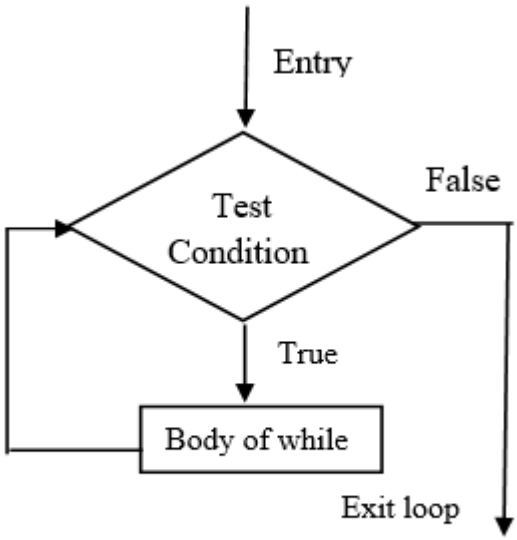


Fig. Flowchart of while loop

Syntax of while-loop

```
while condition :
    body of while loop
```

Example to calculate the sum of first ten numbers

```
num = 10
sum = 0
i = 1
while i <= num:
    sum = sum + i
    i = i + 1
print("Sum of first 10 number is:", sum)
```

Output

```
Sum of first 10 number is: 55
```

Break Statement in Python

The break statement is used inside the loop to exit out of the loop. It is useful when we want to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations. It reduces execution time. Whenever the controller encountered a break statement, it comes out of that loop immediately

Let's see how to break a for a loop when we found a number greater than 5.

Example of using a break statement

```
for num in range(10):
    if num > 5:
        print("stop processing.")
        break
    print(num)
```

Output

```
0
1
2
3
4
5
stop processing.
```

Continue statement in python

The continue statement is used to skip the current iteration and `continue` with the next iteration.

Let's see how to skip a for a loop iteration if the number is 5 and continue executing the body of the loop for other numbers.

Example of a `continue` statement

```
for num in range(3, 8):
    if num == 5:
        continue
    else:
        print(num)
```

Output

```
3
4
6
7
```

Pass statement in Python

The pass is the keyword In Python, which won't do anything. Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.

A pass statement is a Python null statement. When the interpreter finds a pass statement in the program, it returns no operation. Nothing happens when the `pass` statement is executed.

It is useful in a situation where we are implementing new methods or also in exception handling. It plays a role like a placeholder.

Example

```
months = ['January', 'June', 'March', 'April']
for mon in months:
    pass
print(months)
```

Output

```
['January', 'June', 'March', 'April']
```

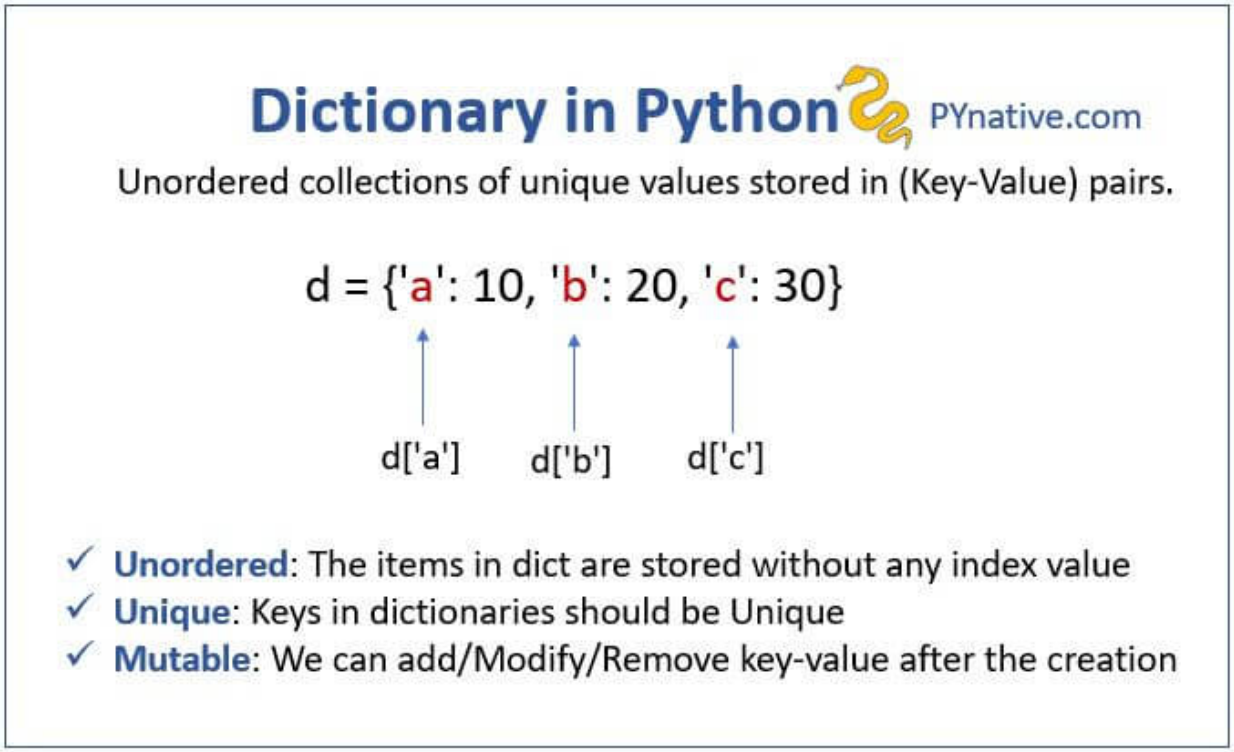
Dictionaries in Python

Dictionaries are ordered collections of unique values stored in (Key-Value) pairs.

In Python version **3.7** and onwards, dictionaries are ordered. In Python **3.6** and earlier, dictionaries are **unordered**.

Python dictionary represents a mapping between a key and a value. In simple terms, a Python dictionary can store pairs of keys and values. Each key is linked to a specific value. Once stored in a dictionary, you can later obtain the value using just the key.

For example, consider the Phone lookup, where it is very easy and fast to find the phone number(value) when we know the name(Key) associated with it.



Characteristics of dictionaries

- **Unordered** (In Python 3.6 and lower version): The items in dictionaries are stored without any index value, which is typically a range of numbers. They are stored as Key-Value pairs, and the keys are their index, which will not be in any sequence.
- **Ordered** (In Python 3.7 and higher version): dictionaries are ordered, which means that the items have a defined order, and that order will not change. A simple Hash Table consists of key-value pair arranged in pseudo-random order based on the calculations from Hash Function.
- **Unique:** As mentioned above, each value has a Key; the Keys in Dictionaries should be unique. If we store any value with a Key that already exists, then the most recent value will replace the old value.
- **Mutable:** The dictionaries are changeable collections, which implies that we can add or remove items after the creation.

Creating a dictionary

There are following three ways to create a dictionary.

- **Using curly brackets**: The dictionaries are created by enclosing the comma-separated Key: Value pairs inside the `{}` curly brackets. The colon '`:`' is used to separate the key and value in a pair.

- **Using `dict()` constructor**: Create a dictionary by passing the comma-separated key: value pairs inside the `dict()`.
- **Using sequence** having each item as a pair (key-value)

Let's see each one of them with an example.

Example:

```
# create a dictionary using {}
person = {"name": "Jessa", "country": "USA", "telephone": 1178}
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# create a dictionary using dict()
person = dict({"name": "Jessa", "country": "USA", "telephone": 1178})
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# create a dictionary from sequence having each item as a pair
person = dict([("name", "Mark"), ("country", "USA"), ("telephone", 1178)])
print(person)

# create dictionary with mixed keys keys
# first key is string and second is an integer
sample_dict = {"name": "Jessa", 10: "Mobile"}
print(sample_dict)
# output {'name': 'Jessa', 10: 'Mobile'}

# create dictionary with value as a list
person = {"name": "Jessa", "telephones": [1178, 2563, 4569]}
print(person)
# output {'name': 'Jessa', 'telephones': [1178, 2563, 4569]}
```

Empty Dictionary

When we create a dictionary without any elements inside the curly brackets then it will be an empty dictionary.

```
emptydict = {}
print(type(emptydict))
# Output class 'dict'
```

Note:

- A dictionary value can be of any type, and duplicates are allowed in that.
- Keys in the dictionary must be unique and of immutable types like string, numbers, or tuples.

Accessing elements of a dictionary

There are two different ways to access the elements of a dictionary.

1. Retrieve value using the key name inside the `[]` square brackets
2. Retrieve value by passing key name as a parameter to the `get()` method of a dictionary.

Example

```
# create a dictionary named person
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# access value using key name in []
print(person['name'])
# Output 'Jessa'

#  get key value using key name in get()
print(person.get('telephone'))
# Output 1178
```

As we can see in the output, we retrieved the value 'Jessa' using key 'name" and value 1178 using its Key 'telephone'.

Get all keys and values

Use the following dictionary methods to retrieve all key and values at once

Method	Description
`keys()`	Returns the list of all keys present in the dictionary.
`values()`	Returns the list of all values present in the dictionary
`items()`	Returns all the items present in the dictionary. Each item will be inside a tuple as a key-value pair.

We can assign each method's output to a separate variable and use that for further computations if required.

Example

```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# Get all keys
print(person.keys())
# output dict_keys(['name', 'country', 'telephone'])
print(type(person.keys()))
# Output class 'dict_keys'

# Get all values
print(person.values())
# output dict_values(['Jessa', 'USA', 1178])
print(type(person.values()))
# Output class 'dict_values'
```

```
# Get all key-value pair
print(person.items())
# output dict_items([('name', 'Jessa'), ('country', 'USA'), ('telephone', 1178)])
print(type(person.items()))
# Output class 'dict_items'
```

Iterating a dictionary

We can iterate through a dictionary using a for-loop and access the individual keys and their corresponding values. Let us see this with an example.

```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# Iterating the dictionary using for-loop
print('key', ':', 'value')
for key in person:
    print(key, ':', person[key])

# using items() method
print('key', ':', 'value')
for key_value in person.items():
    # first is key, and second is value
    print(key_value[0], key_value[1])
```

Output

```
key : value
name : Jessa
country : USA
telephone : 1178

key : value
name Jessa
country USA
telephone 1178
```

Find a length of a dictionary

In order to find the number of items in a dictionary, we can use the `len()` function. Let us consider the personal details dictionary which we created in the above example and find its length.

```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# count number of keys present in  a dictionary
print(len(person))
# output 3
```

Adding items to the dictionary

We can add new items to the dictionary using the following two ways.

- **Using key-value assignment:** Using a simple assignment statement where value can be assigned directly to the new key.
- **Using update() Method:** In this method, the item passed inside the update() method will be inserted into the dictionary. The item can be another dictionary or any iterable like a tuple of key-value pairs.

Now, Let's see how to add two new keys to the dictionary.

Example

```
person = {"name": "Jessa", 'country': "USA", "telephone": 1178}

# update dictionary by adding 2 new keys
person["weight"] = 50
person.update({"height": 6})

# print the updated dictionary
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50,
'height': 6}
```

Note: We can also add more than one key using the update() method.

Example

```
person = {"name": "Jessa", 'country': "USA"}

# Adding 2 new keys at once
# pass new keys as dict
person.update({"weight": 50, "height": 6})
# print the updated dictionary
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'weight': 50, 'height': 6}

# pass new keys as as list of tuple
person.update([("city", "Texas"), ("company", "Google",)])
# print the updated dictionary
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'weight': 50, 'height': 6, 'city':
'Texas', 'company': 'Google'}
```

Modify the values of the dictionary keys

We can modify the values of the existing dictionary keys using the following two ways.

- **Using key name**: We can directly assign new values by using its key name. The key name will be the existing one and we can mention the new value.
- **Using update() method**: We can use the update method by passing the key-value pair to change the value. Here the key name will be the existing one, and the value to be updated will be new.

Example

```
person = {"name": "Jessa", "country": "USA"}

# updating the country name
person["country"] = "Canada"
# print the updated country
print(person['country'])
# Output 'Canada'

# updating the country name using update() method
person.update({"country": "USA"})
# print the updated country
print(person['country'])
# Output 'USA'
```

Removing items from the dictionary

There are several methods to remove items from the dictionary. Whether we want to remove the single item or the last inserted item or delete the entire dictionary, we can choose the method to be used.

Use the following dictionary methods to remove keys from a dictionary.

Method	Description
`pop(key[,d])`	Return and removes the item with the `key` and return its value. If the `key` is not found, it raises `KeyError`.
`popitem()`	Return and removes the last inserted item from the dictionary. If the dictionary is empty, it raises `KeyErr` or.
`del key`	The `del` keyword will delete the item with the key that is passed
`clear()`	Removes all items from the dictionary. Empty the dictionary
`del dict_name`	Delete the entire dictionary

Now, Let's see how to delete items from a dictionary with an example.

Example

```
person = {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50,
'height': 6}

# Remove last inserted item from the dictionary
deleted_item = person.popitem()
print(deleted_item)  # output ('height', 6)
# display updated dictionary
print(person)
```

```
# Output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50}

# Remove key 'telephone' from the dictionary
deleted_item = person.pop('telephone')
print(deleted_item)  # output 1178
# display updated dictionary
print(person)
# Output {'name': 'Jessa', 'country': 'USA', 'weight': 50}

# delete key 'weight'
del person['weight']
# display updated dictionary
print(person)
# Output {'name': 'Jessa', 'country': 'USA'}

# remove all item (key-values) from dict
person.clear()
# display updated dictionary
print(person)  # {}

# Delete the entire dictionary
del person
```

Checking if a key exists

In order to check whether a particular key exists in a dictionary, we can use the `keys()` method and `in` operator. We can use the `in` operator to check whether the key is present in the list of keys returned by the `keys()` method.

In this method, we can just check whether our key is present in the list of keys that will be returned from the `keys()` method.

Let's check whether the 'country' key exists and prints its value if found.

```
person = {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# Get the list of keys and check if 'country' key is present
key_name = 'country'
if key_name in person.keys():
    print("country name is", person[key_name])
else:
    print("Key not found")
# Output country name is USA
```

Copy a Dictionary

We can create a copy of a dictionary using the following two ways

- Using `copy()` method.
- Using the `dict()` constructor

```
dict1 = {'Jessa': 70, 'Emma': 55}

# Copy dictionary using copy() method
dict2 = dict1.copy()
# printing the new  dictionary
print(dict2)
# output {'Jessa': 70, 'Emma': 55}

# Copy dictionary using dict() constructor
dict3 = dict(dict1)
print(dict3)
# output {'Jessa': 70, 'Emma': 55}

# Copy dictionary using the output of items() methods
dict4 = dict(dict1.items())
print(dict4)
# output {'Jessa': 70, 'Emma': 55}
```

Sort dictionary

The built-in method sorted() will sort the keys in the dictionary and returns a sorted list. In case we want to sort the values we can first get the values using the `values()` and then sort them.

Example

```
dict1 = {'c': 45, 'b': 95, 'a': 35}

# sorting dictionary by keys
print(sorted(dict1.items()))
# Output [('a', 35), ('b', 95), ('c', 45)]

# sort dict eys
print(sorted(dict1))
# output ['a', 'b', 'c']

# sort dictionary values
print(sorted(dict1.values()))
# output [35, 45, 95]
```

When to use dictionaries?

Dictionaries are items stored in Key-Value pairs that actually use the mapping format to actually store the values. It uses hashing internally for this. For retrieving a value with its key, the time taken will be very less as `O(1)`.

For example, consider the phone lookup where it is very easy and fast to find the phone number (value) when we know the name (key) associated with it.

So to associate values with keys in a more optimized format and to retrieve them efficiently using that key, later on, dictionaries could be used.