

Dictionaries

Dictionaries are Python's implementation of a data structure, generally known as associative arrays, hashes, or hashmaps.

You can think of a dictionary as a mapping between a set of indexes (known as keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key:value pair or sometimes an item.

You can create a dictionary by placing a comma-separated list of key:value pairs in curly braces { }. Each key is separated from its associated value by a colon :



```
employee1 = {'name': 'Manasseh',  
             'age': 23,  
             'job': 'DevOps',  
             'city': 'Jos',  
             'email': 'manasseh@web.com'}  
  
print(employee1)
```

There are lots of other ways to create a dictionary.

You can use dict() function along with the zip() function, to combine separate lists of keys and values obtained dynamically at runtime.

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains Python code for creating a dictionary from two lists.

```
# Create a dictionary with list of zipped keys/values

keys = ['name', 'age', 'job']
values = ['Bob', 25, 'Dev']
D = dict(zip(keys, values))
print(D)

# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

Important Properties of a Dictionary

Dictionaries are pretty straightforward, but here are a few points you should be aware of when using them.

1. Keys must be unique:
2. A key can appear in a dictionary only once.

Even if you specify a key more than once during the creation of a dictionary, the last value for that key becomes the associated value.



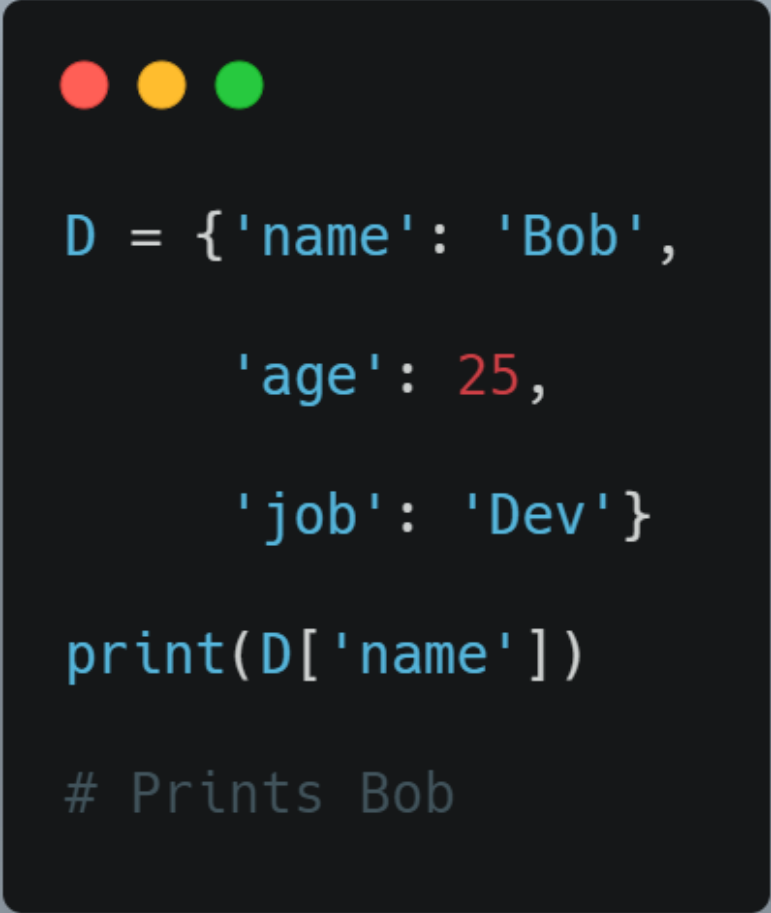
```
D = {'name': 'Bob',  
     'age': 25,  
     'name': 'Jane'}  
  
print(D)  
  
# Prints {'name': 'Jane', 'age': 25}
```

Access Dictionary Items

The order of key:value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.

However, this is not a problem because the items of a dictionary are not indexed with integer indices. Instead, you use the keys to access the corresponding values.

You can fetch a value from a dictionary by referring to its key in square brackets [].



```
D = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
print(D['name'])  
  
# Prints Bob
```

If you refer to a key that is not in the dictionary, you'll get an exception. To avoid such exceptions, you can use the special dictionary `get()` method. This method returns the value for key if key is in the dictionary, else `None`, so that this method never raises a `KeyError`.



```
# When key is present
```

```
print(D.get('name'))
```

```
# Prints Bob
```

```
# When key is absent
```

```
print(D.get('salary'))
```

```
# Prints None
```

Add or Update Dictionary Items.

Adding or updating dictionary items is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, its value is replaced by the new one.



```
D = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
D['name'] = 'Sam'  
  
print(D)  
  
# Prints {'name': 'Sam', 'age': 25, 'job': 'Dev'}
```

If the key is new, it is added to the dictionary with its value.



```
D = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
D['city'] = 'New York'  
  
print(D)  
  
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev',  
          'city': 'New York'}
```

Merge Two Dictionaries

Use the built-in `update()` method to merge the keys and values of one dictionary into another. Note that this method blindly overwrites values of the same key if there's a clash.



```
D1 = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
D2 = {'age': 30,  
      'city': 'New York',  
      'email': 'bob@web.com'}  
  
D1.update(D2)  
  
print(D1)  
  
# Prints {'name': 'Bob', 'age': 30, 'job': 'Dev',  
#        'city': 'New York', 'email': 'bob@web.com'}
```

Remove Dictionary Items

There are several ways to remove items from a dictionary.

1. Remove an Item by Key

If you know the key of the item you want, you can use the `pop()` method. It removes the key and returns its value.



```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
x = D.pop('age')  
  
print(D)  
  
# Prints {'name': 'Bob', 'job': 'Dev'}  
  
# get removed value  
  
print(x)  
  
# Prints 25
```


2. If you don't need the removed value, use the del statement.



```
D = {'name': 'Bob',  
     'age': 25,  
     'job': 'Dev'}  
  
del D['age']  
  
print(D)  
  
# Prints {'name': 'Bob', 'job': 'Dev'}
```

3. Remove all Items

To delete all keys and values from a dictionary, use the `clear()` method.



```
D = {'name': 'Bob',  
      'age': 25,  
      'job': 'Dev'}  
  
D.clear()  
  
print(D)  
  
# Prints {}
```

While Loop

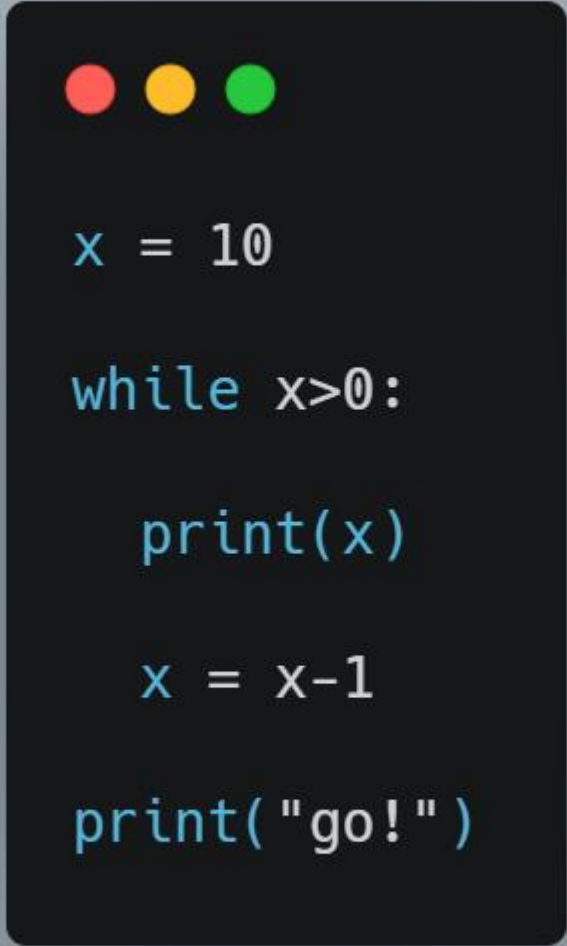
The while loop is somewhat similar to an if statement, it executes the code inside, if the condition is True. However, as opposed to the if statement, the while loop continues to execute the code repeatedly as long as the condition is True. Repeating an action until a condition is met

Syntax:

While condition:

statement(s)

Example, a program for counting down from 10 to 0



```
x = 10

while x>0:

    print(x)

    x = x-1

print( "go!" )
```

So here is what is happening in this example. The loop starts by checking if the condition is met, since x is greater than 0 it prints x which is 10, after that x is reduced by 1, this will continue until the condition is no longer true, i.e when x is less than 0 and it prints go!

The Python Break and Continue Statements

In the above example, the entire body of the while loop is executed on each iteration. Python provides two keywords that terminate a loop iteration prematurely:

The Python break statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.



```
x = 0
```

```
n = 10
```

```
while x <= n:
```

```
    print(x)
```

```
    if x == 4:
```

```
        break
```

```
    x = x + 1
```

```
# prints 0-4
```

In the above example, the program execution is stopped when n is equal to 4.

The Python continue statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted style. It defines a variable n as 5, enters a while loop that runs as long as n is greater than 0. Inside the loop, n is decremented by 1. An if statement checks if n is equal to 2; if true, it executes the continue statement, which jumps the loop back to the condition check. After the if block, n is printed. Once the loop ends, a message 'Loop ended.' is printed.

```
n = 5

while n > 0:

    n -= 1

    if n == 2:

        continue

    print(n)

print('Loop ended.')
```

This time, when n is 2, the `continue` statement causes termination of that iteration. Thus, 2 isn't printed. Execution returns to the top of the loop, the condition is re-evaluated, and it is still true. The loop resumes, terminating when n becomes 0, as previously.