
# Python for DevOps

**Module 3.**

# Lists

# Lists

Lists are used to store multiple items in a single variable (collections of data).
List items are ordered, changeable, and allow duplicate values.

```
fruits = ['apple', 'orange', 'plum']
print(fruits)
['apple', 'orange', 'plum’]

fruits[0]
'apple’
fruits[-1]
'plum'
```

# Change Item Value in the List

```
fruits = ['apple', 'orange', 'plum’]

fruits[1] =  'peach'
print(fruits)
['apple', 'peach', 'plum’]


fruits.insert(2, "watermelon")
print(fruits)
['apple', 'peach', 'watermelon', 'plum’]

fruits.remove('apple')
print(fruits)
['peach', 'watermelon', 'plum']
```

# Examples of operations with Lists

| Example | Output | Explanation |
|---|---|---|
| len([1,2,3]) | 3 | Length of the list |
| [1,2,3]+[4,5,6] | [1,2,3,4,5,6] | Concatenation |
| ["O"]*3 | ["O", "O", "O",] | Multiply |
| 3 in [1,2,3] | True | Check if object in list |
| for x in [1,2,3]:<br>  print x | 1<br>2<br>3 | iterate |

# Built-in List Functions

| Method | Description |
| --- | --- |
| len(list) | Gives the total length of the list. |
| max(list) | Returns item from the list with max value. |
| min(list) | Returns item from the list with min value. |
| any(list) | True if at least one element of an iterable is true |
| all(list) | True if all elements in an iterable are true |
| list(seq) | Converts a tuple into list. |

# List Methods

| Method | Description |
|---|---|
| append() | Adds an element at the end of the list |
| clear() | Removes all the elements from the list |
| copy() | Returns a copy of the list |
| count() | Returns the number of elements with the specified value |
| extend() | Add the elements of a list (or any iterable), to the end of the current list |
| index() | Returns the index of the first element with the specified value |
| insert() | Adds an element at the specified position |
| pop() | Removes the element at the specified position |
| remove() | Removes the first item with the specified value |
| reverse() | Reverses the order of the list |
| sort() | Sorts the list |

# Tuple

# Why tuples?

- Looks like list
- Tuple protected from changes (unwanted and wanted) IMMUTABLE
- We can use tuple as a key for dictianary
- Compact – smaller than list

```
topics = ('lists', 'tuples', 'dictionary')
print(topics)
('lists', 'tuples', 'dictionary')
```

```
topics1 = ('lists', 'tuples', 'dictionary')
topics2 = ['lists', 'tuples', 'dictionary']
topics1.__sizeof__()
48
topics2.__sizeof__()
104
```

# Operations with tuples

```
topics = ('lists', 'tuples', 'dictionary')
topics[1]
'tuples'

topics[1:]
('tuples', 'dictionary')


topics[1] = 'math'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

topics2 = ['lists', 'tuples', 'dictionary']
topics2[1] = 'math'

topics + topics
('lists', 'tuples', 'dictionary', 'lists', 'tuples', 'dictionary')
```

# Examples of operations with tuples

| Example | Output | Explanation |
|---|---|---|
| len((1,2,3)) | 3 | Length of the list |
| (1,2,3)+(4,5,6) | (1,2,3,4,5,6) | Concatenation |
| ("o", "a") *3 | ('o', 'a', 'o', 'a', 'o', 'a') | Multiply |
| 3 in (1,2,3) | True | Check if object in tuple |
| for x in (1,2,3):<br>  print x | 1<br>2<br>3 | iterate |

# Sets

# Sets

Set items are unordered, unchangeable, and do not allow duplicate values.

```
fruits = {"apple", "banana", "cherry"}
print(fruits)
{'apple', 'cherry', 'banana'}
```

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Once a set is created, you cannot change its items, but you can add new items.

# Dictionary

# Dictionaries

Dictionaries are used to store data values in key:value pairs.

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

Dictionaries are:

- Changeable
- No duplicates
- Ordered (you cannot refer to an item by using an index.)

```
mycar = {"brand": "Ford", "color": "Brown"}
print(mycar)
{'brand': 'Ford', 'color': 'Brown'}
```

# Let’s read the value from dictionary

```
mycar = {"brand": "Ford",
         "color": "Brown",
         "model": "Focus",
         "generation": 3}

print(f"My car is {mycar['brand']} {mycar['model']}")
My car is Ford Focus

mycar.keys()
dict_keys(['brand', 'color', 'model', 'generation’])

mycar.values()
dict_values(['Ford', 'Brown', 'Focus', 3])
```

# Change Dictionary Items

```
mycar = {"brand": "Ford",
         "color": "Brown",
         "model": "Focus",
         "generation": 3}

mycar['color'] = 'Red’


del mycar['color']
print(mycar)
{'brand': 'Ford', 'model': 'Focus', 'generation': 3}

mycar.clear() # delete all entries
del mycar  # delete entire dictionary
```

# Fetaures of the keys in Dictionary

Keys shuld be uniq:

```
mypet = {"name": "Bobik", "name": "Sharik"}
print(mypet)
{'name': 'Sharik'}
```

Keys are immutable and could be integers, strings, or tuples,
but not lists, because lists are mutable.

```
mypet = {['name']: "Bobik"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

# Dictionary Methods

| Method | Description |
|---|---|
| clear() | Removes all the elements from the dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| get() | Returns the value of the specified key |
| items() | Returns a list containing a tuple for each key value pair |
| keys() | Returns a list containing the dictionary's keys |
| pop() | Removes the element with the specified key |
| popitem() | Removes the last inserted key-value pair |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| update() | Updates the dictionary with the specified key-value pairs |
| values() | Returns a list of all the values in the dictionary |

# Functions

# Creating and Calling a Function

A function is a block of code which only runs when it is called.

```
def my_shiny_function():
    print("It's my new shiny function")

my_shiny_function()
```

```
def say_hey(name):
    hey = f”Hey {name}!"
    return hey

say_hey('Vasia')
‘Hey Vasia!'
```

# Arguments

Function must get exact same number of arguments that it expects

```
say_hi('Vasia', 'Pete')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: say_hi() takes 1 positional argument but 2 were given
```

```
names = ['Bob', 'Mike', 'Nick']
for name in names:
    say_hi(name)
...
'Hi Bob!'
'Hi Mike!'
'Hi Nick!'
```

# Python Function Arguments: Arbitrary, Keywords and Default

## Default Arguments:

```
def say_hi(name='Vasia’):
    print(f"Hi {name}!!")

say_hi()
Hi Vasia!!
say_hi('Bob')
Hi Bob!!
```

## Keyword Arguments

```
def say_hi(msg, name):
    print(f"{msg} {name} !!")
say_hi(name='Bob', msg ='Hey')
Hey Bob !!
```

## Arbitrary Arguments

```
def say_hi(*name):
    for name in names:
        print(f"Hey {name}")
...
say_hi("Bob", "Mike", "Nick")
Hey Bob
Hey Mike
Hey Nick
```

# Errors and Exceptions

# Syntax Errors VS Exceptions

Statement itself is not correct and the parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected:

```
while true print('helo world')
  File "<stdin>", line 1
    while true print('helo world')
               ^
SyntaxError: invalid syntax
```

If a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it:

```
>>> 123/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

# Handling Exceptions

```
while True:
    try:
        x = int(input("Please enter a number: "))
            break
        except ValueError:
            print("Oops!  That was no valid number.  Try again...")
...
Please enter a number: frgfdg
Oops!  That was no valid number.  Try again...
Please enter a number: 123
>>>
```