Python Tutorial

Sources and further reading

This is a summary of the most used operators and built-in functions in Python. You can find more information at the following links:

https://www.pythoncheatsheet.org/

https://www.w3schools.com/python/

1. Python getting started

1.1. Python Install

Many PCs and Macs will have python already installed. To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

1.2. Python quick start

From there you can write any python code, e.g.

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

Which will write "Hello, World!" in the command line:

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

Furthermore, Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

Let's write our first Python file, called helloworld.py, which can be done in any text editor, e.g. Notepad:

helloworld.py

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

Save your file as “hellpworld.py”. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
exit()
```

1.3. Indentation

Indentation refers to the spaces at the beginning of a code line. Whereas in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Output

```
Five is greater than two!
```

However, without indentation, e.g.

```
if 5 > 2:
print("Five is greater than two!")
```

it returns an error:

```
    print("Five is greater than two!")
        ^
IndentationError: expected an indented block
```

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one. However, you have to use the same number of spaces in the same block of code, otherwise Python will give you an error, e.g.:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

Output

```
    print("Five is greater than two!")
    ^
IndentationError: unexpected indent
```

2. Basic syntax

2.1. Comments

a. Inline comment

```
# This is a comment
```

b. Multiline comment

```
# This is a
# multiline comment
```

c. Code with a comment

```
a = 1  # initialization
```

d. Function Docstring

```
def foo():
    """
    This is a function docstring
    You can also use:
    ''' Function Docstring '''
    """
```

2.2. Math operators

From **highest** to **lowest** precedence:

Operators	**Operation**	**Example**
**	Exponent	2 ** 3 = 8
%	Modulus/Remainder	22 % 8 = 6
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 9
-	Subtraction	5 - 2 = 3
+	Addition	2 + 2 = 4

Examples of expressions:

```
>>> 2 + 3 * 6
# 20

>>> (2 + 3) * 6
# 30

>>> 2 ** 8
#256

>>> 23 // 7
# 3

>>> 23 % 7
# 2

>>> (5 - 1) * ((7 + 1) / (3 - 1))
# 16.0
```

2.3. Augmented Assignment Operators

Operator	**Equivalent**
var += 1	var = var + 1
var -= 1	var = var - 1
var *= 1	var = var * 1
var /= 1	var = var / 1
var //= 1	var = var // 1
var %= 1	var = var % 1
var **= 1	var = var ** 1

Examples:

```
>>> greeting = 'Hello'
>>> greeting += ' world!'
>>> greeting
# 'Hello world!'

>>> number = 1
>>> number += 1
>>> number
# 2

>>> my_list = ['item']
>>> my_list *= 3
>>> my_list
# ['item', 'item', 'item']
```

2.4. Walrus Operator

The Walrus Operator allows assignment of variables within an expression while returning the value of the variable

Example:

```
>>> print(my_var:="Hello World!")
# 'Hello world!'

>>> my_var="Yes"
>>> print(my_var)
# 'Yes'

>>> print(my_var:="Hello")
# 'Hello'
```

2.5. Data Types

Data Type	**Examples**
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

2.6. Concatenation and Replication

String concatenation:

```
>>> 'Alice' 'Bob'
# 'AliceBob'
```

String replication:

```
>>> 'Alice' * 5
# 'AliceAliceAliceAliceAlice'
```

2.7. Variables

You can name a variable anything as long as it obeys the following rules:

a. It can be only one word:

```
>>> # bad
>>> my variable = 'Hello'

>>> # good
>>> var = 'Hello'
```

b. It can use only letters, numbers, and the underscore (_) character:

```
>>> # bad
>>> %$@variable = 'Hello'

>>> # good
>>> my_var = 'Hello'

>>> # good
>>> my_var_2 = 'Hello'
```

c. It can't begin with a number:

```
>>> # this won't work
>>> 23_var = 'hello'
```

d. Variable name starting with an underscore (_) are considered as "unuseful":

```
>>> # _spam should not be used again in the code
>>> _spam = 'Hello'
```

2.8. The `print()` function

The `print()` function writes the value of the argument(s) it is given. It handles multiple arguments, floating point-quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely:

```
>>> print('Hello world!')

# Hello world!

>>> a = 1

>>> print('Hello world!', a)

# Hello world! 1
```

a. The `end` keyword

The keyword argument `end` can be used to avoid the newline after the output, or end the output with a different string:

```
phrase = ['printed', 'with', 'a', 'dash', 'in', 'between']
>>> for word in phrase:
...     print(word, end='-')
...
# printed-with-a-dash-in-between-
```

b. The `sep` keyword

The keyword `sep` specify how to separate the objects, if there is more than one:

```
print('cats', 'dogs', 'mice', sep=',')
# cats,dogs,mice
```

2.9. The `input()` function

This function takes the input from the user and converts it into a string:

```
>>> print('What is your name?')   # ask for their name
>>> my_name = input()
>>> print('Hi, {}'.format(my_name))
# What is your name?
# Martha
# Hi, Martha
```

`input()` can also set a default message without using `print()`:

```
>>> my_name = input('What is your name? ')  # default message
>>> print('Hi, {}'.format(my_name))
# What is your name? Martha
# Hi, Martha
```

It is also possible to use formatted strings (`f'…'`) to avoid using `.format()`:

```
>>> my_name = input('What is your name? ')  # default message
>>> print(f'Hi, {my_name}')
# What is your name? Martha
# Hi, Martha
```

2.10. The `len()` function

Evaluates to the integer value of the number of characters in a string, list, dictionary, etc.:

```
>>> len('hello')
# 5
>>> len(['cat', 3, 'dog'])
# 3
```

Example: Test of emptiness

Test of emptiness of strings, lists, dictionaries, etc., should not use len, but prefer direct boolean evaluation.

```
>>> a = [1, 2, 3]
# bad
>>> if len(a) > 0:  # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
# good
>>> if a: # evaluates to True
...     print("the list is not empty!")
...
# the list is not empty!
```

2.11. The `str()`, `int()` and `float()` functions

These functions allow you to change the type of variable. For example, you can transform from an integer or float to a string:

```
>>> str(29)
# '29'
>>> str(-3.14)
# '-3.14'
```

Or from a string to an integer or float:

```
>>> int('11')
# 11
>>> float('3.14')
# 3.14
```

3. Control flow

Control flow is the order in which individual statements, instructions, or function calls are executed or evaluated. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

3.1. Comparison operators

Operator	**Meaning**
= =	Equal to
!=	Not equal to
<	Less than
>	Greater Than
<=	Less than or Equal to
>=	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
>>> 42 == 42
True
>>> 40 == 42
False
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

3.3. Boolean Operators

There are three Boolean operators: `and`, `or`, and `not`. In the order of precedence, highest to lowest they are `not`, `and` and `or`.

The `and` Operator's *Truth* Table:

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

The `or` Operator's *Truth* Table:

Expression	Evaluates to
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The `not` Operator's *Truth* Table:

Expression	Evaluates to
`not True`	`False`
`not False`	`True`

3.4. Mixing operators

You can mix boolean and comparison operators:

```
>>> (4 < 5) and (5 < 6)
True

>>> (4 < 5) and (9 < 6)
False

>>> (1 == 2) or (2 == 2)
True
```

Also, you can mix use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
>>> # In the statement below 3 < 4 and 5 > 5 gets executed first
evaluating to False
>>> # Then 5 > 4 returns True so the results after True or False is True
>>> 5 > 4 or 3 < 4 and 5 > 5
True
>>> # Now the statement within parentheses gets executed first so True
and False returns False.
>>> (5 > 4 or 3 < 4) and 5 > 5
False
```

3.5. `if` Statements

The `if` statement evaluates an expression, and if that expression is True, it then executes the following indented code:

```
>>> name = 'Debora'

>>> if name == 'Debora':
...     print('Hi, Debora')
...
# Hi, Debora

>>> if name != 'George':
...     print('You are not George')
...
# You are not George
```

The `else` statement executes only if the evaluation of the `if` and all the `elif` expressions are `False`:

```
>>> name = 'Debora'

>>> if name == 'George':
...     print('Hi, George.')
... else:
...     print('You are not George')
...
# You are not George
```

Only after the `if` statement expression is `False`, the `elif` statement is evaluated and executed:

```
>>> name = 'George'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
...
# Hi George!
```

the `elif` and `else` parts are optional.

```
>>> name = 'Antony'

>>> if name == 'Debora':
...     print('Hi Debora!')
... elif name == 'George':
...     print('Hi George!')
... else:
...     print('Who are you?')
...
# Who are you?
```

3.6. Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse, simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, and otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> # this if statement:
>>> if age < 18:
...     print('kid')
... else:
...     print('adult')
...
# output: kid

>>> # is equivalent to this ternary operator:
>>> print('kid' if age < 18 else 'adult')
# output: kid
```

Ternary operators can be chained:

```
>>> age = 15

>>> # this ternary operator:
>>> print('kid' if age < 13 else 'teen' if age < 18 else 'adult')

>>> # is equivalent to this if statement:
>>> if age < 18:
...     if age < 13:
...         print('kid')
...     else:
...         print('teen')
... else:
...     print('adult')
...
# output: teen
```

3.7. Switch-Case Statement

In computer programming languages, a switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via search and map.

a. Matching single values

```
>>> response_code = 201
>>> match response_code:
...     case 200:
...         print("OK")
...     case 201:
...         print("Created")
...     case 300:
...         print("Multiple Choices")
...     case 307:
...         print("Temporary Redirect")
...     case 404:
...         print("404 Not Found")
...     case 500:
...         print("Internal Server Error")
...     case 502:
...         print("502 Bad Gateway")
...
# Created
```

b. Matching with the or Pattern

In this example, the pipe character (`|` or `or`) allows python to return the same response for two or more cases.

```
>>> response_code = 502
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...
# Internal Server Error
```

c. Matching by the length of an Iterable

```
>>> today_responses = [200, 300, 404, 500]
>>> match today_responses:
...     case [a]:
...             print(f"One response today: {a}")
...     case [a, b]:
...             print(f"Two responses today: {a} and {b}")
...     case [a, b, *rest]:
...             print(f"All responses: {a}, {b}, {rest}")
...
# All responses: 200, 300, [404, 500]
```

d. Default value

The underscore symbol (_) is used to define a default case:

```
>>> response_code = 800
>>> match response_code:
...     case 200 | 201:
...         print("OK")
...     case 300 | 307:
...         print("Redirect")
...     case 400 | 401:
...         print("Bad Request")
...     case 500 | 502:
...         print("Internal Server Error")
...     case _:
...         print("Invalid Code")
...
# Invalid Code
```

e. Matching Built-in Classes

```
>>> response_code = "300"
>>> match response_code:
...     case int():
...             print('Code is a number')
...     case str():
...             print('Code is a string')
...     case _:
...             print('Code is neither a string nor a number')
...
# Code is a string
```

e. Guarding Match-Case Statements

```
>>> response_code = 300
>>> match response_code:
...     case int():
...             if response_code > 99 and response_code < 500:
...                 print('Code is a valid number')
...     case _:
...             print('Code is an invalid number')
...
# Code is a valid number
```

3.8. `while` Loop Statements

The while statement is used for repeated execution as long as an expression is `True`:

```
>>> spam = 0
>>> while spam < 5:
...     print('Hello, world.')
...     spam = spam + 1
...
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
# Hello, world.
```

3.9. `break` Statements

If the execution reaches a `break` statement, it immediately exits the `while` loop’s clause:

```
>>> while True:
...     name = input('Please type your name: ')
...     if name == 'your name':
...         break
...
>>> print('Thank you!')
# Please type your name: your name
# Thank you!
```

3.10. `continue` Statements

When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop.

```
>>> while True:
...     name = input('Who are you? ')
...     if name != 'Joe':
...         continue
...     password = input('Password? (It is a fish.): ')
...     if password == 'swordfish':
...         break
...
>>> print('Access granted.')
# Who are you? Charles
# Who are you? Debora
# Who are you? Joe
# Password? (It is a fish.): swordfish
# Access granted.
```

3.11. `for` loop

The `for` loop iterates over a `list`, `tuple`, `dictionary`, `set` or `string`:

```
>>> pets = ['Bella', 'Milo', 'Loki']
>>> for pet in pets:
...     print(pet)
...
# Bella
# Milo
# Loki
```

3.12. The `range()` function

The `range()` function returns a sequence of numbers. It starts from 0, increments by 1, and stops before a specified number:

```
>>> for i in range(5):
...     print(f'Will stop at 5! or 4? ({i})')
...
# Will stop at 5! or 4? (0)
# Will stop at 5! or 4? (1)
# Will stop at 5! or 4? (2)
# Will stop at 5! or 4? (3)
# Will stop at 5! or 4? (4)
```

The `range()` function can also modify its 3 defaults arguments. The first two will be the `start` and `stop` values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
# range(start, stop, step)
>>> for i in range(0, 10, 2):
...    print(i)
...
# 0
# 2
# 4
# 6
# 8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
>>> for i in range(5, -1, -1):
...     print(i)
...
# 5
# 4
# 3
# 2
# 1
# 0
```

3.13. `for else` statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
>>> for i in [1, 2, 3, 4, 5]:
...    if i == 3:
...        break
... else:
...    print("only executed when no item is equal to 3")
```

3.14. Ending a Program with `sys.exit()`

`exit()` function allows exiting Python.

```
>>> import sys

>>> while True:
...     feedback = input('Type exit to exit: ')
...     if feedback == 'exit':
...         print(f'You typed {feedback}.')
...         sys.exit()
...
# Type exit to exit: open
# Type exit to exit: close
# Type exit to exit: exit
# You typed exit
```

4. Functions

4.1. Function arguments

A function can take arguments and return values:

In the following example, the function **`say_hello`** receives the argument “name” and prints a greeting:

```
>>> def say_hello(name):
...     print(f'Hello {name}')
...
>>> say_hello('Carlos')
# Hello Carlos

>>> say_hello('Wanda')
# Hello Wanda

>>> say_hello('Rose')
# Hello Rose
```

4.2. Keyword arguments

To improve code readability, we should be as explicit as possible. We can achieve this in our functions by using Keyword Arguments:

```
>>> def say_hi(name, greeting):
...     print(f"{greeting} {name}")
...
>>> # without keyword arguments
>>> say_hi('John', 'Hello')
# Hello John

>>> # with keyword arguments
>>> say_hi(name='Anna', greeting='Hi')
# Hi Anna
```

4.3. Return values

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The `return` keyword.
- The value or expression that the function should return.

```
>>> def sum_two_numbers(number_1, number_2):
...     return number_1 + number_2
...
>>> result = sum_two_numbers(7, 8)
>>> print(result)
# 15
```

4.4. Global and local scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

```
global_variable = 'I am available everywhere'

>>> def some_function():
...     print(global_variable)  # because is global
...     local_variable = "only available within this function"
...     print(local_variable)
...
>>> # the following code will throw error because
>>> # 'local_variable' only exists inside 'some_function'
>>> print(local_variable)
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
NameError: name 'local_variable' is not defined
```

4.5. The global statement

If you need to modify a global variable from within a function, use the global statement:

```
>>> def spam():
...     global eggs
...     eggs = 'spam'
...
>>> eggs = 'global'
>>> spam()
>>> print(eggs)
#spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

a) If a variable is being used in the global scope (that is, outside all functions), then it is always a global variable.
b) If there is a global statement for that variable in a function, it is a global variable.
c) Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
d) But if the variable is not used in an assignment statement, it is a global variable.

4.6. Lambda function

In Python, a *lambda* function is a single-line, anonymous function, which can have any number of arguments, but it can only have one expression.

Example: This function

```
>>> def add(x, y):
...     return x + y
...
>>> add(5, 3)
# 8
```

is equivalent to the *lambda* function

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
# 8
```

Like regular nested functions, lambdas also work as lexical closures:

```
>>> def make_adder(n):
...     return lambda x: x + n
...
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
# 7
>>> plus_5(4)
# 9
```

5. Data types

Python comes equipped with several built-in data types to help us organize our data. These structures include **lists, dictionaries, tuples and sets.**

5.1. Lists

Lists are one of the 4 data types in Python used to store collections of data.

5.1.1. Getting values with indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0]
# 'table'

>>> furniture[1]
# 'chair'

>>> furniture[2]
# 'rack'

>>> furniture[3]
# 'shelf'
```

5.1.2. Negative indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[-1]
# 'shelf'

>>> furniture[-3]
# 'chair'

>>> f'The {furniture[-1]} is bigger than the {furniture[-3]}'
# 'The shelf is bigger than the chair'
```

5.1.3. Getting sublists with slices

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0:4]
# ['table', 'chair', 'rack', 'shelf']

>>> furniture[1:3]
# ['chair', 'rack']

>>> furniture[0:-1]
# ['table', 'chair', 'rack']

>>> furniture[:2]
# ['table', 'chair']

>>> furniture[1:]
# ['chair', 'rack', 'shelf']

>>> furniture[:]
# ['table', 'chair', 'rack', 'shelf']
```

Slicing the complete list will perform a copy:

```
>>> spam2 = spam[:]
# ['cat', 'bat', 'rat', 'elephant']

>>> spam.append('dog')
>>> spam
# ['cat', 'bat', 'rat', 'elephant', 'dog']

>>> spam2
# ['cat', 'bat', 'rat', 'elephant']
```

5.1.4. Getting a list length with `len()`

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> len(furniture)
# 4
```

5.1.5. Changing values with indexes

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> furniture[0] = 'desk'
>>> furniture
# ['desk', 'chair', 'rack', 'shelf']

>>> furniture[2] = furniture[1]
>>> furniture
# ['desk', 'chair', 'chair', 'shelf']

>>> furniture[-1] = 'bed'
>>> furniture
# ['desk', 'chair', 'chair', 'bed']
```

5.1.6. Concatenation and replication

```
>>> [1, 2, 3] + ['A', 'B', 'C']
# [1, 2, 3, 'A', 'B', 'C']

>>> ['X', 'Y', 'Z'] * 3
# ['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

>>> my_list = [1, 2, 3]
>>> my_list = my_list + ['A', 'B', 'C']
>>> my_list
# [1, 2, 3, 'A', 'B', 'C']
```

5.1.7. Using `for` loops with lists

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> for item in furniture:
...     print(item)
# table
# chair
# rack
# shelf
```

5.1.8. Getting the index in a loop with `enumerate()`

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']

>>> for index, item in enumerate(furniture):
...     print(f'index: {index} - item: {item}')
# index: 0 - item: table
# index: 1 - item: chair
# index: 2 - item: rack
# index: 3 - item: shelf
```

5.1.9. Loop in multiple lists with `zip()`

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> price = [100, 50, 80, 40]

>>> for item, amount in zip(furniture, price):
...     print(f'The {item} costs ${amount}')
# The table costs $100
# The chair costs $50
# The rack costs $80
# The shelf costs $40
```

5.1.10. The `in` and `not` operators

```
>>> 'rack' in ['table', 'chair', 'rack', 'shelf']
# True

>>> 'bed' in ['table', 'chair', 'rack', 'shelf']
# False

>>> 'bed' not in furniture
# True

>>> 'rack' not in furniture
# False
```

5.1.11. The multiple assignment trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> table = furniture[0]
>>> chair = furniture[1]
>>> rack = furniture[2]
>>> shelf = furniture[3]
```

You could type this line of code:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> table, chair, rack, shelf = furniture

>>> table
# 'table'

>>> chair
# 'chair'

>>> rack
# 'rack'

>>> shelf
# 'shelf'
```

The multiple assignment trick can also be used to swap the values in two variables:

```
>>> a, b = 'table', 'chair'
>>> a, b = b, a
>>> print(a)
# chair

>>> print(b)
# table
```

5.1.12. The index method

The `index` method allows you to find the index of a value by passing its name:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.index('chair')
# 1
```

5.1.13. Adding values to a list

a. `Append( )`

`append` adds an element to the end of a list:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.append('bed')
>>> furniture
# ['table', 'chair', 'rack', 'shelf', 'bed']
```

b. `Insert( )`

`insert` adds an element to a list at a given position:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.insert(1, 'bed')
>>> furniture
# ['table', 'bed', 'chair', 'rack', 'shelf']
```

5.1.14. Removing values from a list

a. `del( )`

`del` removes an item using the index:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> del furniture[2]
>>> furniture
# ['table', 'chair', 'shelf']

>>> del furniture[2]
>>> furniture
# ['table', 'chair']
```

b. `remove()`

`remove` removes an item with using actual value of it:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> furniture.remove('chair')
>>> furniture
# ['table', 'rack', 'shelf']
```

Note! Removing repeated items

If the value appears multiple times in the list, only the first instance of the value will be removed.

c. `pop()`

By default, `pop` will remove and return the last item of the list. You can also pass the index of the element as an optional parameter:

```
>>> animals = ['cat', 'bat', 'rat', 'elephant']

>>> animals.pop()
'elephant'

>>> animals
['cat', 'bat', 'rat']

>>> animals.pop(0)
'cat'

>>> animals
['bat', 'rat']
```

5.1.15. Sorting values with `sort()`

```
>>> numbers = [2, 5, 3.14, 1, -7]
>>> numbers.sort()
>>> numbers
# [-7, 1, 2, 3.14, 5]

furniture = ['table', 'chair', 'rack', 'shelf']
furniture.sort()
furniture
# ['chair', 'rack', 'shelf', 'table']
```

You can also pass `True` for the reverse keyword argument to have `sort()` sort the values in reverse order:

```
>>> furniture.sort(reverse=True)
>>> furniture
# ['table', 'shelf', 'rack', 'chair']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the key keyword argument in the `sort()` method call:

```
>>> letters = ['a', 'z', 'A', 'Z']
>>> letters.sort(key=str.lower)
>>> letters
# ['a', 'A', 'z', 'Z']
```

You can use the built-in function `sorted` to return a new list:

```
>>> furniture = ['table', 'chair', 'rack', 'shelf']
>>> sorted(furniture)
# ['chair', 'rack', 'shelf', 'table']
```

5.2. The Tuple data type

5.2.1. Lists vs Tuples

The key difference between tuples and lists is that, while tuples are *immutable* objects, lists are *mutable*. This means that tuples cannot be changed while the lists can be modified. Tuples are more memory efficient than the lists.

```
>>> furniture = ('table', 'chair', 'rack', 'shelf')

>>> furniture[0]
# 'table'

>>> furniture[1:3]
# ('chair', 'rack')

>>> len(furniture)
# 4
```

5.2.2. Converting between `list()` and `tuple()`

```
# Convert list to tuple
>>> tuple(['cat', 'dog', 5])
# ('cat', 'dog', 5)

#Convert tuple to list
>>> list(('cat', 'dog', 5))
# ['cat', 'dog', 5]

#Convert string to list
>>> list('hello')
# ['h', 'e', 'l', 'l', 'o']
```

5.3. Dictionaries

In Python, a dictionary is an *ordered* collection of `key : value` pairs.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a `key:value` pair with del.

Example Dictionary:

```
my_cat = {
    'size': 'fat',
    'color': 'gray',
    'disposition': 'loud'
}
```

5.3.1. Set key, value using subscript operator `[]`

```
>>> my_cat = {
...  'size': 'fat',
...  'color': 'gray',
...  'disposition': 'loud',
... }
>>> my_cat['age_years'] = 2
>>> print(my_cat)
...
# {'size': 'fat', 'color': 'gray', 'disposition': 'loud', 'age_years':
2}
```

5.3.2. Get value using subscript operator `[]`

In case the key is not present in dictionary `KeyError` is raised

```
>>> my_cat = {
...  'size': 'fat',
...  'color': 'gray',
...  'disposition': 'loud',
... }
>>> print(my_cat['size'])
...
# fat
>>> print(my_cat['eye_color'])
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 'eye_color'
```

5.3.3. `values()`

The `values()` method gets the **values** of the dictionary:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for value in pet.values():
...     print(value)
...
# red
# 42
```

5.3.4. `keys()`

The keys() method gets the **keys** of the dictionary:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key in pet.keys():
...     print(key)
...
# color
# age
```

There is no need to use `.keys()` since by default you will loop through keys:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key in pet:
...     print(key)
...
# color
# age
```

5.3.5. `items()`

The `items()` method gets the **items** of a dictionary and returns them as a Tuple:

```
>>> pet = {'color': 'red', 'age': 42}
>>> for item in pet.items():
...     print(item)
...
# ('color', 'red')
# ('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.

```
>>> pet = {'color': 'red', 'age': 42}
>>> for key, value in pet.items():
...     print(f'Key: {key} Value: {value}')
...
# Key: color Value: red
# Key: age Value: 42
```

5.3.6. `get()`

The `get()` method returns the value of an item with the given key. If the key doesn't exist, it returns `None`:

```
>>> wife = {'name': 'Rose', 'age': 33}

>>> f'My wife name is {wife.get("name")}'
# 'My wife name is Rose'

>>> f'She is {wife.get("age")} years old.'
# 'She is 33 years old.'

>>> f'She is deeply in love with {wife.get("husband")}'
# 'She is deeply in love with None'
```

You can also change the default `None` value to one of your choice:

```
>>> wife = {'name': 'Rose', 'age': 33}

>>> f'She is deeply in love with {wife.get("husband", "John")}'
# 'She is deeply in love with Jhon'
```

5.3.7. Adding items with `setdefault()`

It's possible to add an item to a dictionary in this way:

```
>>> wife = {'name': 'Rose', 'age': 33}
>>> if 'has_hair' not in wife:
...     wife['has_hair'] = True
```

Using the `setdefault` method, we can make the same code more short:

```
>>> wife = {'name': 'Rose', 'age': 33}
>>> wife.setdefault('has_hair', True)
>>> wife
# {'name': 'Rose', 'age': 33, 'has_hair': True}
```

5.3.8. Removing items

a. `pop()`

The `pop()` method removes and returns an item based on a given key.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.pop('age')
# 33
>>> wife
# {'name': 'Rose', 'hair': 'brown'}
```

b. `popitem()`

The `popitem()` method removes the last item in a dictionary and returns it.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.popitem()
# ('hair', 'brown')
>>> wife
# {'name': 'Rose', 'age': 33}
```

c. `del()`

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> del wife['age']
>>> wife
# {'name': 'Rose', 'hair': 'brown'}
```

d. `clear()`

The `clear()` method removes all the items in a dictionary.

```
>>> wife = {'name': 'Rose', 'age': 33, 'hair': 'brown'}
>>> wife.clear()
>>> wife
# {}
```

5.3.9. Checking keys in a dictionary

```
>>> person = {'name': 'Rose', 'age': 33}

>>> 'name' in person.keys()
# True

>>> 'height' in person.keys()
# False

>>> 'skin' in person # You can omit keys()
# False
```

5.3.10. Checking values in a dictionary

```
>>>  person = {'name': 'Rose', 'age': 33}

>>> 'Rose' in person.values()
# True

>>> 33 in person.values()
# True
```

5.3.11. Pretty printing

```
>>> import pprint

>>> wife = {'name': 'Rose', 'age': 33, 'has_hair': True, 'hair_color':
'brown', 'height': 1.6, 'eye_color': 'brown'}
>>> pprint.pprint(wife)
# {'age': 33,
#  'eye_color': 'brown',
#  'hair_color': 'brown',
#  'has_hair': True,
#  'height': 1.6,
#  'name': 'Rose'}
```

5.3.12. Merge two dictionaries

```
>>> dict_a = {'a': 1, 'b': 2}
>>> dict_b = {'b': 3, 'c': 4}
>>> dict_c = {**dict_a, **dict_b}
>>> dict_c
# {'a': 1, 'b': 3, 'c': 4}
```

5.4. Sets

The 4th type of data in Python are Sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries.

5.4.1. Initialising a set

There are two ways to create sets: using curly braces {} and the built-in function `set()`

Note! Empty Sets

When creating set, be sure to not use empty curly braces {} or you will get an empty dictionary instead.

```
>>> s = {1, 2, 3}
>>> s = set([1, 2, 3])

>>> s = {}  # this will create a dictionary instead of a set
>>> type(s)
# <class 'dict'>
```

5.4.2. Unordered collections of unique elements

A set automatically remove all the duplicate values.

```
>>> s = {1, 2, 3, 2, 3, 4}
>>> s
# {1, 2, 3, 4}
```

And as an unordered data type, they can't be indexed.

```
>>> s = {1, 2, 3}
>>> s[0]
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: 'set' object does not support indexing
```

5.4.3. Set `add()` and `update()`

Using the `add()` method we can add a single element to the set.

```
>>> s = {1, 2, 3}
>>> s.add(4)
>>> s
# {1, 2, 3, 4}
```

And with `update()`, multiple ones:

```
>>> s = {1, 2, 3}
>>> s.update([2, 3, 4, 5, 6])
>>> s
# {1, 2, 3, 4, 5, 6}
```

5.4.4. Set `remove()` and `discard()`

Both methods will remove an element from the set, but `remove()` will raise a key error if the value doesn't exist.

```
>>> s = {1, 2, 3}
>>> s.remove(3)
>>> s
# {1, 2}

>>> s.remove(3)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# KeyError: 3
```

`discard()` won't raise any errors.

```
>>> s = {1, 2, 3}
>>> s.discard(3)
>>> s
# {1, 2}
>>> s.discard(3)
```

5.4.5. Set `union()`

`union()` or `|` will create a new set with all the elements from the sets provided.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1.union(s2)  # or 's1 | s2'
# {1, 2, 3, 4, 5}
```

5.4.6. Set `intersection()`

`intersection()` or (&) will return a set with only the elements that are common to all of them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s3 = {3, 4, 5}
>>> s1.intersection(s2, s3)  # or 's1 & s2 & s3'
# {3}
```

5.4.7. Set `difference()`

`difference()` or `(-)` will return only the elements that are unique to the first set (invoked set).

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}

>>> s1.difference(s2)  # or 's1 - s2'
# {1}

>>> s2.difference(s1) # or 's2 - s1'
# {4}
```

5.4.8. Set `symmetric_difference()`

`symmetric_difference()` or ^ will return all the elements that are not common between them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.symmetric_difference(s2)  # or 's1 ^ s2'
# {1, 4}
```

6. Python built-in functions

The Python interpreter has a number of functions and types built into it that are always available.

Function	Description
`abs()`	Return the absolute value of a number.
`aiter()`	Return an asynchronous iterator for an asynchronous iterable.
`all()`	Return True if all elements of the iterable are true.
`any()`	Return True if any element of the iterable is true.
`ascii()`	Return a string with a printable representation of an object.
`bin()`	Convert an integer number to a binary string.
`bool()`	Return a Boolean value.
`breakpoint()`	Drops you into the debugger at the call site.
`bytearray()`	Return a new array of bytes.
`bytes()`	Return a new "bytes" object.
`callable()`	Return True if the object argument is callable, False if not.
`chr()`	Return the string representing a character.
`classmethod()`	Transform a method into a class method.
`compile()`	Compile the source into a code or AST object.
`complex()`	Return a complex number with the value real + `imag*1j`.
`delattr()`	Deletes the named attribute, provided the object allows it.
`dict()`	Create a new dictionary.
`dir()`	Return the list of names in the current local scope.

Function	Description
divmod()	Return a pair of numbers consisting of their quotient and remainder.
enumerate()	Return an enumerate object.
eval()	Evaluates and executes an expression.
exec()	This function supports dynamic execution of Python code.
filter()	Construct an iterator from an iterable and returns true.
float()	Return a floating point number from a number or string.
format()	Convert a value to a "formatted" representation.
frozenset()	Return a new frozen set object.
getattr()	Return the value of the named attribute of object.
globals()	Return the dictionary implementing the current module namespace.
hasattr()	True if the string is the name of one of the object's attributes.
hash()	Return the hash value of the object.
help()	Invoke the built-in help system.
hex()	Convert an integer number to a lowercase hexadecimal string.
id()	Return the "identity" of an object.
input()	This function takes an input and converts it into a string.
int()	Return an integer object constructed from a number or string.
isinstance()	Return True if the object argument is an instance of an object.
issubclass()	Return True if class is a subclass of class info.

Function	Description
`iter()`	Return an iterator object.
`len()`	Return the length (the number of items) of an object.
`list()`	Rather than being a function, list is a mutable sequence type.
`locals()`	Update and return a dictionary with the current local symbol table.
`map()`	Return an iterator that applies function to every item of iterable.
`max()`	Return the largest item in an iterable.
`min()`	Return the smallest item in an iterable.
`next()`	Retrieve the next item from the iterator.
`object()`	Return a new featureless object.
`oct()`	Convert an integer number to an octal string.
`open()`	Open file and return a corresponding file object.
`ord()`	Return an integer representing the Unicode code point of a character.
`pow()`	Return base to the power exp.
`print()`	Print objects to the text stream file.
`property()`	Return a property attribute.
`repr()`	Return a string containing a printable representation of an object.
`reversed()`	Return a reverse iterator.
`round()`	Return number rounded to n digits precision after the decimal point.

Function	Description
`set()`	Return a new set object.
`setattr()`	This is the counterpart of `getattr()`.
`slice()`	Return a sliced object representing a set of indices.
`sorted()`	Return a new sorted list from the items in iterable.
`staticmethod()`	Transform a method into a static method.
`str()`	Return a str version of object.
`sum()`	Sums start and the items of an iterable.
`super()`	Return a proxy object that delegates method calls to a parent or sibling.
`tuple()`	Rather than being a function, is actually an immutable sequence type.
`type()`	Return the type of an object.
`vars()`	Return the `dict` attribute for any other object with a `dict` attribute.
`zip()`	Iterate over several iterables in parallel.
`import()`	This function is invoked by the import statement.