

Python Tutorials Notes - Simple Version - 基础版

Machine Learning

Python Tutorials Notes - Simple Version - 基础版

Chapter 1: Python Start

[Python Home](#)

[Python Introduction](#)

Chapter 2: Python Basic Syntax

[Python Comments](#)

[Python Variables](#)

- [1. Create Variable](#)
- [2. Variable Names](#)
- [3. Global Variables](#)

[Python Scope](#)

- [1. Local Scope](#)
- [2. Global Scope](#)

[Python Operators](#)

- [1. Python Arithmetic Operators](#)
- [2. Python Assignment Operators](#)
- [3. Python Comparison Operators](#)
- [4. Python Logical Operators](#)
- [5. Python Identity Operators](#)
- [6. Python Membership Operators](#)
- [7. Python Bitwise Operators](#)

Chapter 3: Python Data Types

[Built-in Data Types](#)

- [1. Setting the Data Type](#)
- [2. Setting the Specific Data Type](#)

[Python Casting](#)

- [1. Specify a Variable Type](#)

[Python Numbers](#)

[Python Strings](#)

- [1. String Literals](#)
- [2. Strings As Arrays](#)

3. String Formatting
4. Escape Character
5. String Methods
6. Reverse A String

Python Booleans

1. Boolean Values

Chapter 4: Python Collections (Arrays)

Python Lists

1. List Operate
2. Items Operate
3. Remove List duplicates

Python Tuples

1. Tuples Operate
2. Items Operate

Python Set

1. Set Operate
2. Item Operate

Python Dictionaries

1. Dict Operate
2. Item Operate

Python Arrays

1. Array Methods

Python Iterators

1. Create an Iterator
2. StopIteration

Chapter 5: Python Logical

Python If ... Else

1. Python Conditions and If statements

Python While Loops

1. Python While Loop

Python For Loops

1. Python For Loops

Chapter 6: Python Object-oriented

Python Functions

Python Lambda

[1. Why Use Lambda Functions](#)

[Python Classes/Objects](#)

[Python Inheritance](#)

[1. Create a Parent Class](#)

[Python Modules](#)

[Chapter 7: Python Handle](#)

[Python Try Except](#)

[1. Exception Handling](#)

[Python Dates](#)

[1. Python Datetime](#)

[2. The strftime\(\) Method](#)

[Python Math](#)

[Python JSON](#)

[Python RegEx](#)

[1. RegEx Functions](#)

[Python PIP](#)

[Python User Input](#)

[Chapter 8: Python File Handling](#)

[Python Open Files](#)

[Python Read Files](#)

[Python Write/Create Files](#)

[1. Write to an Existing File](#)

[2. Create a New File](#)

[Python Delete Files](#)

[Python References - Python 函数参考](#)

[Python Built In Functions](#)

[Python String Methods](#)

[Python List/Array Methods](#)

[Python Dictionary Methods](#)

[Python Tuple Methods](#)

[Python Set Methods](#)

[Python File Methods](#)

[Python Keywords](#)

[Python Built-in Exceptions](#)

[Python Module Reference - Python 常用模块参考](#)

[Python Random Module](#)
[Python Requests Module](#)
[Python statistics Module](#)
[Python Math Module](#)
[Python CMath Module](#)

Chapter 1: Python Start

Python Home

[Python Home](#)

Python Introduction

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.

- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Chapter 2: Python Basic Syntax

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

```
1. # Create Comment
2. # Comments starts with a `#`, and Python will ignore the
   m
3.
4. # this is a comment
5. print("Hello, World!") # this is a comment
6.
7. # print("this is a comment")
8.
9. # Multi Line Comments
```

```
10. # This is a comment
11. # written in
12. # more than just one line
13. print("Hello, World!")
14.
15. """
16. This is a comment
17. written in
18. more than just one line
19. """
20. print("Hello, World!")
21.
22. '''
23. This is a comment
24. written in
25. more than just one line
26. '''
27. print("Hello, World!")
28.
```

Python Variables

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

1. Create Variable

```
1. # Create Variable
2. x = 5
3. y = "John"
4. print(x)
5. print(y)
6.
7. x = 4          # x is of type int
8. x = "Sally"   # x is now of type str
9. print(x)
10.
11. x = "John"
```

```

12. x = 'John' # is the same as
13.
14. x, y, z = "Apple", "Banana", "Cherry"
15. x = y = z = "Orange"
16.
17. # Output Variables
18.
19. x = "awesome"
20. print("Python is " + x)
21.
22. # Join Variables
23. x = "Python is "
24. y = "awesome"
25. z = x + y
26. print(z)
27.
28. x = 5
29. y = 10
30. print(x + y)
31.
32. x = 5
33. y = "John"
34. print(x + y) # Python will give you an error.
35.

```

2. Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

```

1. # Legal variable names:
2. myvar = "John"
3. my_var = "John"
4. _my_var = "John"
5. myVar = "John"

```

```
6. MYVAR = "John"
7. myvar2 = "John"
8.
9. # Illegal variable names:
10. 2myvar = "John"
11. my-var = "John"
12. my var = "John"
```

3. Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

```
1. # Create a variable outside of a function, and use it in
   side the function.
2.
3. x = "awesome"
4.
5. def my_func():
6.     print("Python is " + x)
7.
8. my_func()
9.
10. # Local Variable
11. x = "awesome"
12.
13. def myfunc():
14.     x = "fantastic"
15.     print("Python is " + x)
16.
17. myfunc()
18.
19. print("Python is " + x)
20.
21. # Global Keyword
22. # To create a global variable inside a function, you can
   use the global keyword.
23. def my_func():
24.     global x
```



```

25. x = "fantastic"
26.
27. my_func()
28.
29. print("Python is " + x)
30.
31. # Change global variable
32. x = "awesome"
33.
34. def myfunc():
35.     global x
36.     x = "fantastic"
37.
38. myfunc()
39.
40. print("Python is " + x)
41.

```

Python Scope

A variable is only available from inside the region it is created. This is called **scope**.

1. Local Scope

A variable created inside a function belongs to the **local scope** of that function, and can only be used inside that function.

```

1.
2. def my_func():
3.     x = 300
4.     print(x)
5.
6. my_func() # A variable created inside a function is available inside that function.
7.
8. # Function Inside Function
9. def my_func():
10.     x = 300
11.     def my_innerfunc():
12.         print(x)
13.     my_innerfunc()

```

```
14.  
15. my_func()  
16.
```

2. Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

```
1. # A variable created outside of a function is global and  
   can be used by anyone.  
2. def myfunc():  
3.     print(x)  
4.  
5. myfunc()  
6.  
7. print(x)  
8.  
9. # Naming Variables  
10. x = 300  
11.  
12. def my_func():  
13.     x = 200  
14.     print(x)  
15.  
16. my_func()  
17.  
18. print(x)  
19.  
20. # Global Keyword  
21. # If you need to create a global variable, but are stuck  
   in the local scope, you can use the global keyword.  
22. # The global keyword makes the variable global.  
23.  
24. def my_func():  
25.     global x  
26.     x = 300  
27.  
28. my_func()
```

```
29.  
30. print(x)  
31.  
32. # To change the value of a global variable inside a func  
    tion, refer to the variable by using the global keyword  
33.  
34. x = 300  
35.  
36. def my_func():  
37.     global x  
38.     x = 200  
39.  
40. my_func()  
41.  
42. print(x)  
43.
```

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

1. Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y

/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

2. Python Assignment Operators

Assignment operators are used to assign values to variables

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
^=	x ^= 3	x = x ^ 3
=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

* |=; x |= 3; x = x | 3 *

3. Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y

!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

4. Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

5. Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

6. Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

7. Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Description	Example
&	AND	Sets each bit to 1 if both bits are 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

* | ; OR ; Sets each bit to 1 if one of two bits is 1 *

Chapter 3: Python Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type: `str`

Numeric Types: `int`, `float`, `complex`

Sequence Types: `list`, `tuple`, `range`

Mapping Type: `dict`

Set Types: `set`, `frozenset`

Boolean Type: `bool`

Binary Types: `bytes`, `bytearray`, `memoryview`

```
1. x = 5
2. print(type(x)) # get the data type of any object by using the type() function
```

1. Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name": "John", "age": 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x =	bytearray(5)
x = memoryview(bytes(5))	memoryview

2. Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list

x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Python Casting

1.Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole - number), or a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```

1. # Integer
2. x = int(1)      # 1
3. y = int(2.8)    # 2
4. z = int("3")    # 3
5.
6. # Float
7. x = float(1)     # 1.0
8. y = float(2.8)   # 2.8

```



```
9. z = float("3")    # 3.0
10. w = float("4.2") # 4.2
11.
12. # Strings
13. x = str("s1")     # 's1'
14. y = str(2)        # '2'
15. z = str(3.0)      # '3.0'
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

```
1. # Create Number
2. x = 1    # int
3. y = 2.8  # float
4. z = 1j   # complex
5.
6. # Get the type of any Object
7. type(x)
8.
9. # Integer
10. x = 1
11. y = 35673444324353
12. z = -3232421
13. # Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
14.
15. # Float
16. x = 1.10
17. y = 1.0
18. z = -35.59    # Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
19. x = 35e3
20. y = 13E4
21. z = -87.7e100 # Float can also be scientific numbers with an "e" to indicate the power of 10.
```

```
22.
23.
24. # Complex
25. x = 3+5j
26. y = 5j
27. z = -5j # Complex numbers are written with a "j" as the
    imaginary part.
28.
29. # Type Conversion
30. a = float(x)
31. b = int (y)
32. c = complex(x) # You can convert from one type to another
    with the int(), float(), and complex() methods.
33. **Note: You cannot convert complex numbers into another
    number type.**
34.
35. # Random Number
36. # Python does not have a random() function to make a random
    number, but Python has a built-in module called random
    that can be used to make random numbers.
37.
38. import random
39. print(random.randrange(1, 10)) # Import the random module,
    and display a random number between 1 and 9.
40.
```

Python Strings

1. String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

```
1. # Print String
2. print("Hello")
3. print('Hello')
4. a = 'Hello'
5. b = "Hello"
6. print(a, b)
```

```
7.
8. a = ""
9. Lorem ipsum dolor sit amet, consectetur adipiscing elit,

10. sed do eiusmod tempor incididunt
11. ut labore et dolore magna aliqua.
12. ""      # three double quotes
13.
14. a = '''
15. Lorem ipsum dolor sit amet,consectetur adipiscing elit,
16. sed do eiusmod tempor incididunt
17. ut labore et dolore magna aliqua.
18. '''     # three single quotes
19.
```

2. Strings As Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

```
1. # Index
2. a = "Hello, World!"
3. x = a[1]
4.
5. # Slicing
6. a[2:5]
7. a[-5:-2]
8.
9. # length of String
10. len(a)
11.
12. # Check String
13. if 'he' in a:
14.     print(True)
15.
16. if 'oo' not in a:
17.     print(False)
```

```
18.  
19. # String Join  
20. a = "Hello"  
21. b = "World"  
22. c = a + b  
23. c = a + " " + b  
24.  
25. # String methods  
26. a.strip()  
27. a.lower()  
28. a.upper()  
29. a.replace("H", "J")  
30. a.split(",")  
31.  
32.
```

3. String Formatting

To make sure a string will display as expected, we can format the result with the `format()` method.

The `format()` method allows you to format selected parts of a string.

Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?

To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:

```
1. # String Format  
2. txt = "The price is {} dollars"  
3. print(txt.format(price)) # Use the format() method to i  
   insert numbers into strings  
4.  
5. txt = "The price is {:.2f} dollars"  
6.  
7.  
8. # Multiple Values  
9. quantity = 3  
10. item = 567  
11. price = 49.95
```

```

12. order = "I want {} pieces of item {} for {:.2f} dollars."
13. print(order.format(quantity, item, price))
14.
15. # Index Numbers
16. quantity = 3
17. item = 567
18. price = 49.95
19. order = "I want to pay {2:.2f} dollars for {0} pieces of
    item {1}."
20. print(order.format(quantity, item, price))
21.
22. name = "John"
23. txt = "His name is {1}. {1} is {0} years old."
24. print(txt.format(age, name)) # refer to the same value more than once, use the index number
25.
26. # Named Indexes
27. order = "I have a {car_name}, it is a {model}."
28. print(order.format(car_name = "Ford", model = "Mustang"))
29.

```

4. Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash `\` followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```

1. txt = "We are the so-called \"Vikings\" from the north."

```

Other escape characters used in Python:

Code	Result
<code>\'</code>	Single Quote
<code>\</code>	Backslash

\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

5. String Methods

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found

isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list

rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

6. Reverse A String

Learn how to reverse a String in Python.

There is no built-in function to reverse a String in Python.

The fastest (and easiest?) way is to use a slice that steps backwards, -1.

```

1. txt = "Hello World"[::-1]
2. print(txt)
3.
4. # Create a slice that starts at the end of the string, a
   nd moves backwards.
5.
6. # In this particular example, the slice statement [::-1]
   means start at the end of the string and end at positio
   n 0, move with the step -1, negative one, which means on
   e step backwards.
7.
8.
9.
```

Python Booleans

Booleans represent one of two values: `True` or `False`.

1. Boolean Values

In programming you often need to know if an expression is `True` or `False`.

You can evaluate any expression in Python, and get one of two answers, `True` or `False`.

When you compare two values, the expression is evaluated and Python returns the Boolean answer

```
1.
2. print(10 > 9)
3. print(10 == 9)
4. print(10 < 9)
5.
6. # Conditions
7. a = 200
8. b = 33
9.
10. if b > a:
11.     print("b is greater than a")
12. else:
13.     print("b is not greater than a")
14.
15. # Evaluate Values and Variables
16. bool("Hello")
17. bool(15)
18. x = "Hello"
19. y = 5
20. bool(x)
21. bool(y) # The bool() function allows you to evaluate an
            y value, and give you True or False in return.
22.
23. # Most Values are True
24. # Almost any value is evaluated to True if it has some s
    ort of content.
25. # Any string is True, except empty strings.
26. # Any number is True, except 0.
27. # Any list, tuple, set, and dictionary are True, except
    empty ones.
```

```
28.
29. bool("abc")
30. bool(123)
31. bool(["apple", "cherry", "banana"]) # all will return True
32.
33. # Some Values are False
34. In fact, there are not many values that evaluates to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.
35.
36. bool(False)
37. bool(None)
38. bool(0)
39. bool("")
40. bool(())
41. bool([])
42. bool({}) # all will return False
43.
44. # One more value, or object in this case, evaluates to False, and that is if you have an object that is made from a class with a __len__ function that returns 0 or False.
45.
46. class my_class():
47.     def __len__(self):
48.         return 0
49.
50. my_obj = my_class()
51. print(bool(my_obj))
52.
53. # Functions can Return a Boolean
54. def myFunction():
55.     return True
56.
57. print(myFunction())
58.
59.
60. def myFunction():
61.     return True
```

```
62.  
63. if myFunction():  
64.     print("YES!")  
65. else:  
66.     print("NO!")  
67.  
68. # built-in functions  
69. x = 200  
70. print(isinstance(x, int))  
71.
```

Chapter 4: Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python Lists

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

List Methods:

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

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 item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

1. List Operate

```

1.
2. # Create List
3. list = ['apple', 'banana', 'cherry']
4. list = list(('apple', 'banana', 'cherry'))
5. a = 'is'
6. list = ['apple','boy', a, 2]
7. list = [[2,3,4], 2, 3, [22, 3,3]]
8.
9.
10. # Copy List
11. mylist = list.copy() # make a copy of a list with the co
    py() method:
12.
13. mylist = list(list) # make a copy of a list with the lis
    t() method:
14.
15. mylist = list * 2
16.
17. list  = [1, 2, 3, 4]
18. mylist = list ** 2
19.
20. # Check List
21. if list > 4:
22.     print(True)
23.

```

```

24. # Join List
25. list1 = ['a', 'b', 'c']
26. list2 = [1, 2, 3]
27.
28. list3 = list1 + list2
29.
30. # appending one by one
31. for x in list2:
32.     list1.append(x)
33.
34. list.extend(list2)
35.
36. # Loop List
37. list = ["apple", "banana", "cherry", "orange", "kiwi",
           "melon", "mango"]
38.
39. length = len(list) # print the number of items in this
    list:
40.
41. for x in list:
42.     print(x)
43.
44. for i, x in enumerate(list):
45.     print(i, x)
46.
47. for i in range(len(list)):
48.     print(i, list[i])
49.
50. # Built-in List
51.
52. list.index('apple') # Get the index of an item
53. list.count('apple') # Count an item
54. list.reverse()      # Reverse the list
55. list.sort()         # Sort the list
56.

```

2. Items Operate

```

1. # Check if 'apple' is present in the list
2. if 'apple' in list:
3.     print(True)

```

```

4.
5. # Add Item
6. list.append('orange')
7. list.insert(1, 'orange')
8.
9. # Remove Item
10. list[2:5] = []
11. list[:] = []
12. list.remove('orange')
13. list.pop()
14. del list[0]
15. del list # delete the list completely
16. list.clear()
17.
18. # Access Item
19. list[1] # the first item has index 0, using index number
20. list[-1] # using negative indexing
21. list[2:5] # using range of indexes, index 2(included) a
    nd end at index 5 (not included)
22. list[:4]
23. list[2:]
24. list[-4:-1]
25.
26. list[1][0]
27. list[1][:2] # subset lists of lists
28.
29. # Change Item
30. list[1] = 'orange' # replace the value
31.

```

3. Remove List duplicates

Learn how to remove duplicates from a List in Python.

```

1. # Remove any duplicates from a List
2. my_list = ["a", "b", "a", "c", "c"]
3. my_list = list(dict.fromkeys(my_list))
4. print(my_list)
5.
6. # Function
7. def my_function(x):

```

```
8.     return list(dict.fromkeys(x))
9.
10. mylist = my_function(["a", "b", "a", "c", "c"])
11.
12. print(mylist)
13.
```

Python Tuples

A tuple is a collection which is **ordered** and **unchangeable**. In Python tuples are written with round brackets.

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

1. Tuples Operate

```
1. # Create Tuple
2. tuple = ('apple', 'banana', 'cherry')
3.
4. tuple = ("apple", ) # One item tuple, remember the comma
5.
6. my_tuple = tuple(("apple", "banana", "cherry"))
7.
8. # Loop Tuple
9. for x in tuple:
10.     print(x)
11.
12. # Length of Tuple
13. len(tuple)
14.
15. # Join Tuple
16.
17. tuple1 = ("a", "b", "c")
```

```
18. tuple2 = (1, 2, 3)
19. tuple3 = tuple1 + tuple2
20.
```

2. Items Operate

```
1. # Access Item
2. tuple[1]
3.
4. tuple[-1]
5.
6. tuple[2:5]
7.
8. tuple[-4:-1]
9.
10. # Change Item
11. x = ("apple", "banana", "cherry")
12. y = list(x)
13. y[1] = "kiwi"
14. x = tuple(y)
15.
16. # Check Item
17. if 'apple' in tuple:
18.     print("Yes, 'apple' is in the fruits tuple.")
19.
20. # Add Item
21. **Once a tuple is created, you cannot add items to it. T
    uples are unchangeable.**
22.
23. # Remove Item
24. **Tuples are unchangeable, so you cannot remove items fr
    om it, but you can delete the tuple completely**
25.
26. del tuple # `del` keyword can delete the tuple complete
    ly
27.
28. # Built-in Methods
29. count('apple') # Returns the number of times a specifi
    ed value occurs in a tuple
30. index('apple') # Searches the tuple for a specified va
    lue and returns the position of where it was found
```


Python Set

A set is a collection which is unordered and unindexed. In Python, sets are written with curly brackets.

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all the elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	Removes the items in this set that are also included in another, specified set
<code>discard()</code>	Remove the specified item
<code>intersection()</code>	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>	Returns whether two sets have a intersection or not
<code>issubset()</code>	Returns whether another set contains this set or not
<code>issuperset()</code>	Returns whether this set contains another set or not
<code>pop()</code>	Removes an element from the set
<code>remove()</code>	Removes the specified element

<code>symmetric_difference()</code>	Returns a set with the symmetric differences of two sets
<code>symmetric_difference_update()</code>	inserts the symmetric differences from this set and another
<code>union()</code>	Return a set containing the union of sets
<code>update()</code>	Update the set with the union of this set and others

1. Set Operate

```
1. # Create Set
2. set = {"apple", "banana", "cherry"}
3.
4. my_set = set(('apple', 'banana', 'cherry'))
5.
6. # Note: Sets are unordered, so you cannot be sure in which order the items will appear.
7.
8. # Loop Set
9. for x in set:
10.     print(x)
11.
12. # Length of Set
13. len(set)
14.
15. # Join Set
16. set1 = {'a', 'b', 'c'}
17. set2 = {1, 2, 3}
18. set3 = set1.union(set2) # The `union()` method returns a new set with all items from both sets.
19.
20. set1.update(set2) # The `update()` method inserts the items in set2 into set1.
21. # Note: Both union() and update() will exclude any duplicate items.
22.
23.
24.
```

2. Item Operate

```
1.
2. # Check Item
3. if 'apple' in set:
4.     print(True)
5.
6. # Change Item
7. **Note: Sets are unordered, so you cannot be sure in which order the items will appear.**
8.
9. # Add Item
10. set.add("orange")
11.
12. set.update(['orange', 'mango', 'grapes'])
13.
14. # Remove Item
15. set.remove('apple')
16. # Note: If the item to remove does not exist, remove() will raise an error.
17.
18. set.discard('apple')
19. # Note: If the item to remove does not exist, discard() will NOT raise an error.
20.
21. x = set.pop()
22. # Note: Sets are unordered, so when using the pop() method, you will not know which item that gets removed.
23.
24. set.clear() # empties the set
25.
26. del set # 'del' keyword will delete the set completely
27.
```

Python Dictionaries

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

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

1. Dict Operate

```
1. # Create Dict
2. dict = {
3.     "brand" : "Ford",
4.     "model" : "Mustang"
5.     "year" : 1984
6. }
7.
8. # Loop Dict
9. for key in dict:
10.     print(key) # Print all key names in the dictionary,
11.               # one by one
12.
13. for key in dict:
14.     print(dict[key]) # Print all values in the dictionary,
15.                   # one by one
```

```
15. for val in dict.values():
16.     print(val) # You can also use the values() method to
    return values of a dictionary
17.
18. for key, val in dict.items():
19.     print(key, val) # Loop through both keys and values,
    by using the items() method
20.
21. # Length of Dict
22. len(dict)
23.
24.
25. # Copy Dict
26. dict2 = dict1 # You cannot copy a dictionary simply by
    typing dict2 = dict1, because: dict2 will only be a ref
    erence to dict1, and changes made in dict1 will automati
    cally also be made in dict2.
27.
28. my_dict = dict.copy() # Make a copy of a dictionary with
    the copy() method
29.
30. my_dict = dict(dict)
31.
32. my_dict = dict(brand="Ford", model="Mustang", year=1964)
33.
34. # Nested Dict
35.
36. my_dict = {
37.     "child1" : {
38.         "name" : "Emil",
39.         "year" : 2004
40.     },
41.     "child2" : {
42.         "name" : "Tobias",
43.         "year" : 2007
44.     },
45.     "child3" : {
46.         "name" : "Linus",
47.         "year" : 2011
48.     }
49. }
```

```
50.
51. child1 = {
52.     "name" : "Emil",
53.     "year" : 2004
54. }
55. child2 = {
56.     "name" : "Tobias",
57.     "year" : 2007
58. }
59. child3 = {
60.     "name" : "Linus",
61.     "year" : 2011
62. }
63.
64. my_dict = {
65.     "child1" : child1,
66.     "child2" : child2,
67.     "child3" : child3
68. }
69.
```

2. Item Operate

```
1. # Access Item
2. x = dict["model"]
3.
4. x = dict.get("model") # There is also a method called ge
   t() that will give you the same result.
5.
6. # Change Item
7. dict["year"] = 2020
8.
9. # Check if Key Exists
10. if 'model' in dict:
11.     print("Yes, 'model' is one of the keys in the dict d
       ictionary")
12.
13. # Add Item
14.
15. dict['color'] = 'red'
16.
```

```
17. # Remove Item
18.
19. dict.pop('model') # The pop() method removes the item with the specified key name
20.
21. dict.popitem() # The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead)
22.
23. del dict['model'] # The del keyword removes the item with the specified key name
24.
25. del dict # The del keyword can also delete the dictionary completely
26.
27. dict.clear() # The del keyword can also delete the dictionary completely
28.
```

Python Arrays

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

Arrays

- Note: This page shows you how to use LISTS as ARRAYS, however, to work with arrays in Python you will have to import a library, like the NumPy library.

```
1. # Create Array
2. cars = ["Ford", "Volvo", "BMW"]
3.
4. # Access Element
5. cars[0]
6.
7. # Modify Element
8. cars[0] = "Toyota"
9.
10. # Length of An Array
11. x = len(cars) # Note: The length of an array is always one more than the highest array index.
```

```

12.
13. # Loop Array
14. for x in cars:
15.     print(x)
16.
17. # Add Element
18. cars.append("Honda")
19.
20. # Remove Array
21. cars.pop(1)
22.
23. cars.remove("Volvo") # Note: The list's remove() method
    only removes the first occurrence of the specified value.
24.

```

1. Array Methods

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

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 item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

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

Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator

```
1. # Return an iterator from a tuple, and print each value
2. my_tuple = ("apple", "banana", "cherry")
3. my_it = iter(my_tuple)
4.
5. print(next(my_it))
6. print(next(my_it))
7. print(next(my_it))
8.
9. # Strings are also iterable objects, containing a sequence of characters
10. mystr = "banana"
11. myit = iter(mystr)
12.
13. print(next(myit))
14. print(next(myit))
15. print(next(myit))
16. print(next(myit))
17. print(next(myit))
18. print(next(myit))
19.
20. # Looping Through an Iterator
21. # Iterate the values of a tuple
22. mytuple = ("apple", "banana", "cherry")
```

```
23.  
24. for x in mytuple:  
25.     print(x)  
26.  
27. # Iterate the characters of a string  
28. mystr = "banana"  
29.  
30. for x in mystr:  
31.     print(x)
```

1. Create an Iterator

To create an object/class as an iterator you have to implement the methods

`__iter__()` and `__next__()` to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

```
1. class MyNumbers:  
2.     def __iter__(self):  
3.         self.a = 1  
4.         return self  
5.  
6.     def __next__(self):  
7.         x = self.a  
8.         self.a += 1  
9.         return x  
10.  
11. myclass = MyNumbers()  
12. myiter = iter(myclass)  
13.  
14. print(next(myiter))  
15. print(next(myiter))  
16. print(next(myiter))  
17. print(next(myiter))
```

```
18. print(next(myiter))
19.
```

2. StopIteration

The example above would continue forever if you had enough `next()` statements, or if it was used in a `for` loop.

To prevent the iteration to go on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times.

```
1. class MyNumbers:
2.     def __iter__(self):
3.         self.a = 1
4.         return self
5.
6.     def __next__(self):
7.         if self.a <= 20:
8.             x = self.a
9.             self.a += 1
10.            return x
11.        else:
12.            raise StopIteration
13.
14. myclass = MyNumbers()
15. myiter = iter(myclass)
16.
17. for x in myiter:
18.     print(x)
```

Chapter 5: Python Logical

Python If ... Else

1. Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`

- Not Equals: $a \neq b$
- Less than: $a < b$
- Less than or equal to: $a \leq b$
- Greater than: $a > b$
- Greater than or equal to: $a \geq b$

These conditions can be used in several ways, most commonly in “if statements” and loops.

```
1. # If Statement
2.
3. a = 33
4. b = 200
5. if b > a:
6.     print("b is greater than a ")
7.
8. if a > b: print("a is greater than b") # short Hand if
9.
10. # Indentation
11. a = 33
12. b = 200
13. if b > a:
14. print("b is greater than a") # you will get an error
15.
16. # Elif
17. a = 33
18. b = 33
19. if b > a:
20.     print("b is greater than a")
21. elif a == b:
22.     print("a and b are equal")
23.
24. # Else
25. a = 200
26. b = 33
27. if b > a:
28.     print("b is greater than a")
29. elif a == b:
30.     print("a and b are equal")
31. else:
32.     print("a is greater than b")
```

```
33.
34.
35. print("A") if a > b else print("B") # Short Hand If ...
    Else
36.
37. # This technique is known as Ternary Operators(三元运算符), or Conditional Expressions.
38.
39. print("A") if a > b else print("=") if a == b else print
    ("B")
40.
41. # And
42. a = 200
43. b = 33
44. c = 500
45. if a > b and c > a:
46.     print("Both conditions are True")
47.
48. # Or
49. a = 200
50. b = 33
51. c = 500
52. if a > b or a > c:
53.     print("At least one of the conditions is True")
54.
55. # Nested If
56.
57. x = 41
58.
59. if x > 10:
60.     print("Above ten,")
61.     if x > 20:
62.         print("and also above 20!")
63.     else:
64.         print("but not above 20.")
65.
66. # The pass Statement
67. a = 33
68. b = 200
69. if b > a:
70.     pass
```

```
71. # if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.
72.
```

Python While Loops

Python has two primitive loop commands:

- while loops
- for loops

1. Python While Loop

With the while loop we can execute a set of statements as long as a condition is true.

```
1. i = 1
2. while i < 6:
3.     print(i)
4.     i +=1 # Note: remember to increment i, or else the
           loop will continue forever.
5.
6. # Break Statement
7. i = 1
8. while i < 6:
9.     print(i)
10.    if i == 3:
11.        break # the break statement we can stop the loop even if the while condition is true
12.    i += 1
13.
14. # Continue Statement
15. i = 0
16. while i < 6:
17.    i += 1
18.    if i == 3:
19.        continue # `continue` statement we can stop the current iteration, and continue with next.
20.    print(i)
21.
22. # Else Statement
23. i = 1
```

```
24. while i < 6:
25.     print(i)
26.     i += 1
27. else:      # With the else statement we can run a block
               of code once when the condition no longer is true.
28.     print("i is no longer less than 6")
29.
```

Python For Loops

1. Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
1. # For - List
2. fruits = ["apple", "banana", "cherry"]
3. for x in fruits:
4.     print(x)
5.
6. # For - String
7. for x in "banana":
8.     print(x)
9.
10. # Break Statement
11. fruits = ["apple", "banana", "cherry"]
12. for x in fruits:
13.     print(x)
14.     if x == "banana":
15.         break
16.
17. fruits = ["apple", "banana", "cherry"]
18. for x in fruits:
19.     if x == "banana":
20.         break
```

```
21.     print(x)
22.
23. # Continue Statement
24.
25. fruits = ["apple", "banana", "cherry"]
26. for x in fruits:
27.     if x == "banana":
28.         continue
29.     print(x)
30.
31.
32. # For - Range
33. # To loop through a set of code a specified number of ti
    mes, we can use the range() function,
34. # The range() function returns a sequence of numbers, st
    arting from 0 by default, and increments by 1 (by defaul
    t), and ends at a specified number.
35.
36. for x in range(6):
37.     print(x)
38.
39. # Note that range(6) is not the values of 0 to 6, but th
    e values 0 to 5.
40. for x in range(2, 6):
41.     print(x)
42.
43. # The range() function defaults to increment the sequenc
    e by 1, however it is possible to specify the increment
    value by adding a third parameter: range(2, 30, 3)
44. for x in range(2, 30, 3):
45.     print(x)
46.
47. # Else in For Loop
48. for x in range(6):
49.     print(x)
50. else:
51.     print("Finally finished!")
52.
53.
54. # Nested Loops
55. adj = ["red", "big", "tasty"]
```



```
56. fruits = ["apple", "banana", "cherry"]
57.
58. for x in adj:
59.     for y in fruits:
60.         print(x, y)
61.
62. # Pass Statement
63. for x in [0, 1, 2]:
64.     pass
65.
```

Chapter 6: Python Object-oriented

Python Functions

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

You can pass data, known as parameters, into a function.

A function can return data as a result.

```
1. # Creating a Function
2. def my_func():          # In Python a function is defined using the def keyword
3.     print("Hello")
4.
5. # Calling a Function
6. def my_function():
7.     print("Hello from a function")
8.
9. my_function()
10.
11. # Arguments
12. def my_function(fname):
13.     print(fname + " Refsnes")
14.
15. my_function("Emil")
16. my_function("Tobias")
17. my_function("Linus")
18.
```

```
19. # Arguments are often shortened to args in Python docume
    ntations.
20.
21. # Parameters or Arguments
22.
23. # Number of Arguments
24. def my_function(fname, lname):
25.     print(fname + " " + lname)
26.
27. my_function("Emil", "Refsnes")
28.
29. def my_function(fname, lname):
30.     print(fname + " " + lname)
31.
32. my_function("Emil")    # will get an error
33.
34. # Arbitrary Arguments, *args
35. def my_function(*kids):
36.     print("The youngest child is " + kids[2])
37.
38. my_function("Emil", "Tobias", "Linus")
39.
40. # Keyword Arguments
41. def my_function(child3, child2, child1):
42.     print("The youngest child is " + child3)
43.
44. my_function(child1 = "Emil", child2 = "Tobias", child3 =
    "Linus")
45.
46. # Arbitrary Keyword Arguments, **kwargs
47. def my_function(**kid):
48.     print("His last name is " + kid["lname"])
49.
50. my_function(fname = "Tobias", lname = "Refsnes")
51.
52. # Default Parameter Value
53. def my_function(country = "Norway"):
54.     print("I am from " + country)
55.
56. my_function("Sweden")
57. my_function("India")
```

```
58. my_function()
59. my_function("Brazil")
60.
61. # Passing a List as an Argument
62. def my_function(food):
63.     for x in food:
64.         print(x)
65.
66. fruits = ["apple", "banana", "cherry"]
67.
68. my_function(fruits)
69.
70. # Return Values
71. def my_function(x):
72.     return 5 * x
73.
74. print(my_function(3))
75. print(my_function(5))
76. print(my_function(9))
77.
78. # The pass Statement
79. def myfunction():
80.     pass
81.
82. # Recursion 递归
83. def tri_recursion(k):
84.     if(k > 0):
85.         result = k + tri_recursion(k - 1)
86.         print(result)
87.     else:
88.         result = 0
89.     return result
90.
91. print("\n\nRecursion Example Results")
92. tri_recursion(6)
93.
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

```
1.  
2. x = lambda a: a + 10  
3. print(x(5))  
4.  
5. x = lambda a, b : a * b  
6. print(x(5, 6))  
7.  
8. x = lambda a, b, c : a + b + c  
9. print(x(5, 6, 2))  
10.
```

1. Why Use Lambda Functions

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number.

```
1. def my_func(n):  
2.     return lambda a : a * n  
3.  
4. my_doubler = my_func(2)  
5.  
6. print(my_doubler(11))  
7.  
8.  
9. def my_func(n):  
10.    return lambda a : a * n  
11.  
12. my_doubler = my_func(2)
```

```
13. my_trippler = my_func(3)
14.
15. print(my_doubler(11))
16. print(my_trippler(11))
17.
```

Python Classes/Objects

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a “blueprint” for creating objects.

```
1. # Create a Class
2. class MyClass:
3.     x = 5
4.
5. # Create Object
6. p1 = MyClass()
7. print(p1.x)
```

The `__init__()` Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in **init()** function.

All classes have a function called **init()**, which is always executed when the class is being initiated.

Use the **init()** function to assign values to object properties, or other operations that are necessary to do when the object is being created.

```
1. # Create a class, use the __init__() function to assign
   values for name and age.
2. class Person:
3.     def __init__(self, name, age):
4.         self.name = name
```

```
5.     self.age = age
6.
7. p1 = Person("John", 36)
8. print(p1.name)
9. print(p1.age)
10.
11. **Note: The __init__() function is called automatically
    every time the class is being used to create a new object.**
12.
13. # Object Methods
14. # Objects can also contain methods. Methods in objects are
    functions that belong to the object.
15.
16. class Person:
17.     def __init__(self, name, age):
18.         self.name = name
19.         self.age = age
20.
21.     def my_func(self):
22.         print("Hello my name is " + self.name)
23.
24. p1 = Person("John", 36)
25. p1.my_func()
26.
27. **Note: The `self` parameter is a reference to the current
    instance of the class, and is used to access variables
    that belong to the class.**
28.
29. # The self Parameter
30. # Use the words mysillyobject and abc instead of self
31. def __init__(mysillyobject, name, age):
32.     mysillyobject.name = name
33.     mysillyobject.age = age
34.
35. def my_func(abc):
36.     print("Hello my name is " + abc.name)
37.
38. p1 = Person("John", 36)
39. p1.my_func()
40.
```

```
41. # Modify Object Properties
42. p1.age = 40
43.
44. # Delete Object Properties
45. del p1.age
46.
47. # Delete Objects
48. del p1
49.
50. # The pass Statement
51. class Person:
52.     pass
53.
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

1. Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class.

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method.

```
1. class Person:
2.     def __init__(self, fname, lname):
3.         self.firstname = fname
4.         self.lastname = lname
5.
6.     def print_name(self):
7.         print(self.firstname, self.lastname)
8.
9. # Use the Person class to create an object, and then execute the printname method:
10.
11. x = Person("John", "Doe")
```

```
12. x.print_name()
13.
14. # Create a Child Class
15. class Student(Person):
16.     pass
17.
18. x = Student("Mike", "Olsen")
19. x.print_name()
20.
21. # Add the __init__() Function
22. class Student(Person):
23.     def __init__(self, fname, lname):
24.         #add properties etc.
25.
26. # Note: The child's __init__() function overrides the in
    heritance of the parent's __init__() function.
27.
28. class Student(Person):
29.     def __init__(self, fname, lname):
30.         Person.__init__(self, fname, lname)
31. # To keep the inheritance of the parent's __init__() fun
    ction, add a call to the parent's __init__() function
32.
33. # Use the super() Function
34. class Student(Person):
35.     def __init__(self, fname, lname):
36.         super().__init__(fname, lname)
37.
38. # Add Properties
39. class Student(Person):
40.     def __init__(self, fname, lname):
41.         super().__init__(fname, lname)
42.         self.graduationyear = 2019
43.
44. class Student(Person):
45.     def __init__(self, fname, lname, year):
46.         super().__init__(fname, lname)
47.         self.graduationyear = year
48.
49. x = Student("Mike", "Olsen", 2019)
50.
```



```

51.
52. # Add Methods
53. class Student(Person):
54.     def __init__(self, fname, lname, year):
55.         super().__init__(fname, lname)
56.         self.graduationyear = year
57.
58.     def welcome(self):
59.         print("Welcome", self.firstname, self.lastname, "to
        the class of", self.graduationyear)
60.
61. # If you add a method in the child class with the same n
    ame as a function in the parent class, the inheritance o
        f the parent method will be overridden.
62.

```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

```

1. # Create a module
2. def greeting(name):    # Save this code in a file named m
    ymodule.py
3.     print("Hello, " + name)
4.
5. # Use a Module
6. import mymodule
7.
8. mymoudle.greeting("Jonathan")
9. # Note: When using a function from a module, use the syn
    tax: module_name.function_name.
10.
11. # Variables in Module
12. # The module can contain functions, as already describe
    d, but also variables of all types (arrays, dictionarie
    s, objects etc).
13.
14. person1 = {    # Save this code in the file mymodule.py

```

```
15.     "name": "John",
16.     "age": 36,
17.     "country": "Norway"
18. }
19.
20. import mymodule
21.
22. a = mymodule.person1["age"]
23. print(a)
24.
25. # Naming a Module
26. # You can name the module file whatever you like, but it
    must have the file extension .py
27.
28. # Re-naming a Module
29. import mymodule as mx
30. a = mx.person1("age")
31. print(a)
32.
33. # Built-in Modules
34. import platform
35. x = platform.system()
36. print(x)
37.
38. # Using the dir() Function
39. # There is a built-in function to list all the function
    names (or variable names) in a module. The dir() functi
    on.
40.
41. import platform
42. x = dir(platform)
43. print(x)
44.
45. # Note: The dir() function can be used on all modules, a
    lso the ones you create yourself.
46.
47. # Import From Module
48. You can choose to import only parts from a module, by us
    ing the from keyword.
49.
50. def greeting(name):
```

```
51.     print("Hello, " + name)
52.
53. person1 = {
54.     "name" : "John",
55.     "age"   : 36,
56.     "country" : "Norway"
57. }
58.
59. from mymodule import person1
60.
61. print (person1["age"])
62.
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]` , not `-- mymodule.person1["age"]--`

Chapter 7: Python Handle

Python Try Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

1. Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement.

```
1.
2. # try block
3. try:
4.     print(x)
5. except:
6.     print("An Exception occurred")
7.
8. # Many Exceptions
```

```
9. try:
10.     print(x)
11. except NameError:
12.     print("Variable x is not defined")
13. except:
14.     print("Something else went wrong")
15.
16. # Else
17. try:
18.     print("Hello")
19. except:
20.     print("Something went wrong")
21. else:
22.     print("Nothing went wrong")
23.
24. # Finally
25. try:
26.     print(x)
27. except:
28.     print("Something went wrong")
29. finally:
30.     print("The 'try except' is finished")
31.
32. try:
33.     f = open("demofile.txt")
34.     f.write("Lorum Ipsum")
35. except:
36.     print("Something went wrong when writing to the file")
37. finally:
38.     f.close()
39.
40. # Raise an exception
41. x = -1
42.
43. if x < 0:
44.     raise Exception("Sorry, no numbers below zero")
45.
46. # Raise a TypeError if x is not an integer
47. x = "hello"
48.
49. if not type(x) is int:
```

```
50.     raise TypeError("Only integers are allowed")
51.
```

Python Dates

1. Python Datetime

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

```
1. import datetime
2. x = datetime.datetime.now()
3. print(x)    # 2020-10-16 11:54:42.839588
4.
5. print(x.year)
6. print(x.strftime("%A"))
7.
8. # Creating Date Objects
9. x = datetime.datetime(2020, 5, 17)
10. print(x)
11.
12. # The strftime() Method
13. x = datetime.datetime(2018, 6, 1)
14. print(x.strftime("%B"))
15.
```

2. The strftime() Method

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec

%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%

Python Math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

```
1. x = min(5, 10, 25)
2. y = max(5, 10, 25)
3. x = abs(-7.25)
4. x = pow(4, 3)
5.
6. # math module
7. import math
8. x = math.sqrt(64)
9. x = math.ceil(1.4)
10. y = math.floor(1.4)
11. x = math.pi
12.
```

Python JSON

- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.

```
1. import json
2.
3. # Parse JSON - Convert from JSON to Python
4.
5. x = '{ "name":"John", "age":30, "city":"New York"}' # some JSON:
6. y = json.loads(x) # parse x:
7. print(y["age"]) # the result is a Python dictionary:
8.
9. # Convert from Python to JSON
10. # a Python object (dict):
11. x = {
12.     "name": "John",
13.     "age": 30,
14.     "city": "New York"
15. }
16.
17. # convert into JSON:
18. y = json.dumps(x)
19.
20. # the result is a JSON string:
21. print(y)
```

```

22.
23. # You can convert Python objects of the following types,
    into JSON strings:
24.
25. - dict
26. - list
27. - tuple
28. - string
29. - int
30. - float
31. - True
32. - False
33. - None
34.
35. print(json.dumps({"name": "John", "age": 30}))
36. print(json.dumps(["apple", "bananas"]))
37. print(json.dumps(("apple", "bananas")))
38. print(json.dumps("hello"))
39. print(json.dumps(42))
40. print(json.dumps(31.76))
41. print(json.dumps(True))
42. print(json.dumps(False))
43. print(json.dumps(None))
44.

```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true

False	false
None	null

```
1. x = {
2.     "name": "John",
3.     "age": 30,
4.     "married": True,
5.     "divorced": False,
6.     "children": ("Ann","Billy"),
7.     "pets": None,
8.     "cars": [
9.         {"model": "BMW 230", "mpg": 27.5},
10.        {"model": "Ford Edge", "mpg": 24.1}
11.    ]
12. }
13.
14. print(json.dumps(x))
15.
16. # Format the Result
17. json.dumps(x, indent=4)
18.
19. json.dumps(x, indent=4, separators=(". ", " = "))
20.
21. # Order the Result
22. json.dumps(x, indent=4, sort_keys=True)
23.
```

Python RegEx

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

```
1. # RegEx Module
2. import re
3.
4. # RegEx in Python
5. txt = "The rain in Spain"
```

```
6. x = re.search("^The.*Spain$", txt)
7.
```

1. RegEx Functions

The re module offers a set of functions that allows us to search a string for a match.

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

```
1. import re
2.
3. # findall()
4. txt = "The rain in Spain"
5. x = re.findall("ai", txt)
6. print(x)
7.
8. txt = "The rain in Spain"
9. x = re.findall("Portugal", txt)
10. print(x) # Return an empty list if no match was found
11.
12. # search()
13. txt = "The rain in Spain"
14. x = re.search("\s", txt)
15.
16. print("The first white-space character is located in position:", x.start())
17.
18. txt = "The rain in Spain"
19. x = re.search("Portugal", txt)
20. print(x)
21.
22. # split()
23. txt = "The rain in Spain"
24. x = re.split("\s", txt)
```

```

25. print(x)
26.
27. txt = "The rain in Spain"
28. x = re.split("\s", txt, 1)
29. print(x)
30.
31. # sub()
32. txt = "The rain in Spain"
33. x = re.sub("\s", "9", txt)
34. print(x)
35.
36. txt = "The rain in Spain"
37. x = re.sub("\s", "9", txt, 2)
38. print(x)
39.
40.

```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value None will be returned, instead of the Match Object.

```

1. txt = "The rain in Spain"
2. x = re.search("ai", txt)
3. print(x) #this will print an object
4.
5. The Match object has properties and methods used to retrieve information about the search, and the result:
6.
7. # .span() returns a tuple containing the start-, and end positions of the match.
8. # .string returns the string passed into the function
9. # .group() returns the part of the string where there was a match
10.
11. txt = "The rain in Spain"
12. x = re.search(r"\bS\w+", txt)
13. print(x.span()) # (12, 17)
14.
15. txt = "The rain in Spain"

```

```
16. x = re.search(r"\bS\w+", txt)
17. print(x.string)
18.
19. txt = "The rain in Spain"
20. x = re.search(r"\bS\w+", txt)
21. print(x.group())
22.
23. Note: If there is no match, the value None will be returned, instead of the Match Object.
```

Python PIP

What is PIP?

PIP is a package manager for Python packages, or modules if you like.

Note: If you have Python version 3.4 or later, PIP is included by default.

What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

```
1. # Check if PIP is Installed
2. >>> pip --version
3.
4. # Install PIP
5.
6. # Download a Package
7. >>> pip install camelcase
8.
9. # Using a package
10. import camelcase
11. c = camelcase.Camelcase()
12. txt = "Hello, World"
13. print(c.hum(txt))
14.
15. # Find Packages
16.
17. # Remove Packages
18. >>> uninstall
19.
20. >>> pip uninstall camelcase
```

```
21.  
22. # List packages  
23.  
24. >>> pip list  
25.  
26. Package            Version  
27. -----  
28. camelcase           0.2  
29. mysql-connector     2.1.6  
30. pip                 18.1  
31. pymongo             3.6.1  
32. setuptools         39.0.1  
33.
```

Python User Input

User Input

Python allows for user input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

The following example asks for the username, and when you entered the username, it gets printed on the screen:

```
1. # Python 3.6  
2. username = input("Enter username: ")  
3. print("Username is: " + username)  
4.  
5. # Python 2.7  
6. username = raw_input("Enter username:")  
7. print("Username is: " + username)  
8.
```

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

Chapter 8: Python File Handling

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

Python Open Files

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist

`"a"` - Append - Opens a file for appending, creates the file if it does not exist

`"w"` - Write - Opens a file for writing, creates the file if it does not exist

`"x"` - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

`"t"` - Text - Default value. Text mode

`"b"` - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
1. f = open("demofile.txt")
2. # The code above is the same as:
3.
4. f = open("demofile.txt", "rt")
5. # Because "r" for read, and "t" for text are the default
   values, you do not need to specify them.
```

Note: Make sure the file exists, or else you will get an error.

Python Read Files

Open a File on the Server

demofile.txt

Hello! Welcome to demofile.txt This file is for testing purposes. Good Luck!

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file

```
1. f = open("demofile.txt", "r")
2. print(f.read())
3.
4. # Open a file on a different location
5. f = open("D:\\myfiles\\welcome.txt", "r")
6. print(f.read())
7.
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return.

```
1.
2. # Return the 5 first characters of the file
3. f = open("demofile.txt", "r")
4. print(f.read(5))
5.
6.
7.
```

Python Write/Create Files

1. Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

“a” - Append - will append to the end of the file

“w” - Write - will overwrite any existing content

```

1.
2. # appending
3. f = open("demofile2.txt", "a")
4. f.write("Now the file has more content!")
5. f.close()
6.
7. #open and read the file after the appending:
8. f = open("demofile2.txt", "r")
9. print(f.read())
10.
11. # overwrite
12. f = open("demofile3.txt", "w")
13. f.write("Woops! I have deleted the content!")
14. f.close()
15.
16. #open and read the file after the appending:
17. f = open("demofile3.txt", "r")
18. print(f.read())
19.
20. # Note: the "w" method will overwrite the entire file.
21.

```

2. Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

`"x"` - Create - will create a file, returns an error if the file exist

`"a"` - Append - will create a file if the specified file does not exist

`"w"` - Write - will create a file if the specified file does not exist

```

1. # Create a file called "myfile.txt".
2. f = open("myfile.txt", "x")
3.
4. # Create a new file if it does not exist
5. f = open("myfile.txt", "w")

```

Python Delete Files

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function

```
1. import os
2. os.remove("demo_file.txt")
3.
4. # Check if File exit
5. if os.path.exists("demofile.txt"):
6.     os.remove("demofile.txt")
7. else:
8.     print("The file does not exist")
9.
10. # Delete Folder
11. os.rmdir("myfolder") # To delete an entire folder, use the os.rmdir() method.
12.
```

Note: You can only remove empty folders.

Python References - Python 函数参考

Python Built In Functions

Python has a set of built-in functions.

Function	Description
✓ abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
✓ bool()	Returns the boolean value of the specified object

bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
✓ complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
✓ dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
✓ float()	Returns a floating point number
✓ format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object

help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
✓ input()	Allowing user input
✓ int()	Returns an integer number
✓ isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
✓ len()	Returns the length of an object
✓ list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
✓ max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
✓ min()	Returns the smallest item in an iterable
✓ next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
✓ open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
✓ pow()	Returns the value of x to the power of y
✓ print()	Prints to the standard output device

property()	Gets, sets, deletes a property
✓ range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
✓ round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
@staticmethod()	Converts a method into a static method
✓ str()	Returns a string object
✓ sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
✓ tuple()	Returns a tuple
✓ type()	Returns the type of an object
vars()	Returns the dict property of an object
zip()	Returns an iterator, from two or more iterators

Python String Methods

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case

casefold()	Converts string into lower case
center()	Returns a centered string
✓ count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
✓ endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
✓ format()	Formats specified values in a string
format_map()	Formats specified values in a string
✓ index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
✓ isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
✓ islower()	Returns True if all characters in the string are lower case
✓ isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
✓ istitle()	Returns True if the string follows the rules of a title
✓ isupper()	Returns True if all characters in the string are upper case
✓ join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string

✓ lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
✓ replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
✓ split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
✓ strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
✓ title()	Converts the first character of each word to upper case
translate()	Returns a translated string
✓ upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Note: All string methods returns new values. They do not change the original string.

Python List/Array Methods

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

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 item with the specified value
✓ reverse()	Reverses the order of the list
✓ sort()	Sorts the list

Python Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

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

Python Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
✓ count()	Returns the number of times a specified value occurs in a tuple
✓ index()	Searches the tuple for a specified value and returns the position of where it was found

Python Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
✓ add()	Adds an element to the set
✓ clear()	Removes all the elements from the set
✓ copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set

✓ discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
✓ pop()	Removes an element from the set
✓ remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
✓ union()	Return a set containing the union of sets
✓ update()	Update the set with the union of this set and others

Python File Methods

Python has a set of methods available for the file object.

Method	Description
✓ close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective

flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
✓ read()	Returns the file content
readable()	Returns whether the file stream can be read or not
✓ readline()	Returns one line from the file
✓ readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
✓ write()	Writes the specified string to the file
✓ writelines()	Writes a list of strings to the file

Python Keywords

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers.

Method	Description
✓ and	A logical operator
✓ as	To create an alias
assert	For debugging
✓ break	To break out of a loop
✓ class	To define a class

✓ continue	To continue to the next iteration of a loop
✓ def	To define a function
✓ del	To delete an object
✓ elif	Used in conditional statements, same as else if
✓ else	Used in conditional statements
✓ except	Used with exceptions, what to do when an exception occurs
✓ False	Boolean value, result of comparison operations
✓ finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
✓ for	To create a for loop
✓ from	To import specific parts of a module
✓ global	To declare a global variable
✓ if	To make a conditional statement
✓ import	To import a module
✓ in	To check if a value is present in a list, tuple, etc.
✓ is	To test if two variables are equal
✓ lambda	To create an anonymous function
✓ None	Represents a null value
nonlocal	To declare a non-local variable
✓ not	A logical operator
✓ or	A logical operator
✓ pass	A null statement, a statement that will do nothing

✓ raise	To raise an exception
✓ return	To exit a function and return a value
✓ True	Boolean value, result of comparison operations
✓ try	To make a try...except statement
✓ while	To create a while loop
✓ with	Used to simplify exception handling
✓ yield	To end a function, returns a generator

Python Built-in Exceptions

Built-in Exceptions

The table below shows built-in exceptions that are usually raised in Python.

Method	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an “end of file” condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indentation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary

KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete
LookupError	Raised when errors raised cant be found
MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the method
OSError	Raised when a system related operation causes an error
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific expectations
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces
SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined
UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a unicode problem occurs
UnicodeEncodeError	Raised when a unicode encoding problem occurs
UnicodeDecodeError	Raised when a unicode decoding problem occurs
UnicodeTranslateError	Raised when a unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

Python Module Reference - Python 常用模块参考

Python Random Module

Python has a built-in module that you can use to make random numbers.

The `random` module has a set of methods.

Method	Description
<code>seed()</code>	Initialize the random number generator
<code>getstate()</code>	Returns the current internal state of the random number generator
<code>setstate()</code>	Restores the internal state of the random number generator
<code>getrandbits()</code>	Returns a number representing the random bits
<code>randrange()</code>	Returns a random number between the given range
<code>randint()</code>	Returns a random number between the given range
<code>choice()</code>	Returns a random element from the given sequence
<code>choices()</code>	Returns a list with a random selection from the given sequence
<code>shuffle()</code>	Takes a sequence and returns the sequence in a random order
<code>sample()</code>	Returns a given sample of a sequence
<code>random()</code>	Returns a random float number between 0 and 1
<code>uniform()</code>	Returns a random float number between two given parameters
<code>triangular()</code>	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters
<code>betavariate()</code>	Returns a random float number between 0 and 1 based on the Beta distribution (used in statistics)

expovariate()	Returns a random float number based on the Exponential distribution (used in statistics)
gammavariate()	Returns a random float number based on the Gamma distribution (used in statistics)
gauss()	Returns a random float number based on the Gaussian distribution (used in probability theories)
lognormvariate()	Returns a random float number based on a log-normal distribution (used in probability theories)
normalvariate()	Returns a random float number based on the normal distribution (used in probability theories)
vonmisesvariate()	Returns a random float number based on the von Mises distribution (used in directional statistics)
paretovariate()	Returns a random float number based on the Pareto distribution (used in probability theories)
weibullvariate()	Returns a random float number based on the Weibull distribution (used in statistics)

Python Requests Module

```
1. import requests
2. x = requests.get('https://w3schools.com/python/demopage.
   htm')
3. print(x.text)
4.
```

Definition and Usage

The requests module allows you to send HTTP requests using Python.

The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

Download and Install the Requests Module

```
1. # Navigate your command line to the location of PIP, and
   type the following.
2. >>> pip install requests
3.
4. # syntax
5. requests.methodname(params)
```

Methods

Method	Description
<code>delete(url, args)</code>	Sends a DELETE request to the specified url
<code>get(url, params, args)</code>	Sends a GET request to the specified url
<code>head(url, args)</code>	Sends a HEAD request to the specified url
<code>patch(url, data, args)</code>	Sends a PATCH request to the specified url
<code>post(url, data, json, args)</code>	Sends a POST request to the specified url
<code>put(url, data, args)</code>	Sends a PUT request to the specified url
<code>request(method, url, args)</code>	Sends a request of the specified method to the specified url

Python statistics Module

Python statistics Module

Python has a built-in module that you can use to calculate mathematical statistics of numeric data.

The statistics module was new in Python 3.4.

Statistics Methods

Method	Description
<code>statistics.harmonic_mean()</code>	Calculates the harmonic mean (central location) of the given data

statistics.mean()	Calculates the mean (average) of the given data
statistics.median()	Calculates the median (middle value) of the given data
statistics.median_grouped()	Calculates the median of grouped continuous data
statistics.median_high()	Calculates the high median of the given data
statistics.median_low()	Calculates the low median of the given data
statistics.mode()	Calculates the mode (central tendency) of the given numeric or nominal data
statistics.pstdev()	Calculates the standard deviation from an entire population
statistics.stdev()	Calculates the standard deviation from a sample of data
statistics.pvariance()	Calculates the variance of an entire population
statistics.variance()	Calculates the variance from a sample of data

Python Math Module

Python math Module

Python has a built-in module that you can use for mathematical tasks.

The `math` module has a set of methods and constants.

Math Methods

Method	Description
math.acos()	Returns the arc cosine of a number
math.acosh()	Returns the inverse hyperbolic cosine of a number
math.asin()	Returns the arc sine of a number
math.asinh()	Returns the inverse hyperbolic sine of a number
math.atan()	Returns the arc tangent of a number in radians
math.atan2()	Returns the arc tangent of y/x in radians

<code>math.atanh()</code>	Returns the inverse hyperbolic tangent of a number
<code>math.ceil()</code>	Rounds a number up to the nearest integer
<code>math.comb()</code>	Returns the number of ways to choose k items from n items without repetition and order
<code>math.copysign()</code>	Returns a float consisting of the value of the first parameter and the sign of the second parameter
<code>math.cos()</code>	Returns the cosine of a number
<code>math.cosh()</code>	Returns the hyperbolic cosine of a number
<code>math.degrees()</code>	Converts an angle from radians to degrees
<code>math.dist()</code>	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point
<code>math.erf()</code>	Returns the error function of a number
<code>math.erfc()</code>	Returns the complementary error function of a number
<code>math.exp()</code>	Returns E raised to the power of x
<code>math.expm1()</code>	Returns $E^x - 1$
<code>math.fabs()</code>	Returns the absolute value of a number
<code>math.factorial()</code>	Returns the factorial of a number
<code>math.floor()</code>	Rounds a number down to the nearest integer
<code>math.fmod()</code>	Returns the remainder of x/y
<code>math.frexp()</code>	Returns the mantissa and the exponent, of a specified number
<code>math.fsum()</code>	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
<code>math.gamma()</code>	Returns the gamma function at x
<code>math.gcd()</code>	Returns the greatest common divisor of two integers
<code>math.hypot()</code>	Returns the Euclidean norm
<code>math.isclose()</code>	Checks whether two values are close to each other, or not

<code>math.isfinite()</code>	Checks whether a number is finite or not
<code>math.isinf()</code>	Checks whether a number is infinite or not
<code>math.isnan()</code>	Checks whether a value is NaN (not a number) or not
<code>math.isqrt()</code>	Rounds a square root number downwards to the nearest integer
<code>math.ldexp()</code>	Returns the inverse of <code>math.frexp()</code> which is $x * (2^{**}i)$ of the given numbers x and i
<code>math.lgamma()</code>	Returns the log gamma value of x
<code>math.log()</code>	Returns the natural logarithm of a number, or the logarithm of number to base
<code>math.log10()</code>	Returns the base-10 logarithm of x
<code>math.log1p()</code>	Returns the natural logarithm of $1+x$
<code>math.log2()</code>	Returns the base-2 logarithm of x
<code>math.perm()</code>	Returns the number of ways to choose k items from n items with order and without repetition
<code>math.pow()</code>	Returns the value of x to the power of y
<code>math.prod()</code>	Returns the product of all the elements in an iterable
<code>math.radians()</code>	Converts a degree value into radians
<code>math.remainder()</code>	Returns the closest value that can make numerator completely divisible by the denominator
<code>math.sin()</code>	Returns the sine of a number
<code>math.sinh()</code>	Returns the hyperbolic sine of a number
<code>math.sqrt()</code>	Returns the square root of a number
<code>math.tan()</code>	Returns the tangent of a number
<code>math.tanh()</code>	Returns the hyperbolic tangent of a number
<code>math.trunc()</code>	Returns the truncated integer parts of a number

Math Constants

Constant	Description
math.e	Returns Euler's number (2.7182...)
math.inf	Returns a floating-point positive infinity
math.nan	Returns a floating-point NaN (Not a Number) value
math.pi	Returns PI (3.1415...)
math.tau	Returns tau (6.2831...)

Python CMath Module

Python cmath Module

Python has a built-in module that you can use for mathematical tasks for complex numbers.

The methods in this module accepts `int`, `float`, and `complex` numbers. It even accepts Python objects that has a `__complex__()` or `__float__()` method.

The methods in this module almost always return a complex number. If the return value can be expressed as a real number, the return value has an imaginary part of 0.

The `cmath` module has a set of methods and constants.

cmath Methods

Method	Description
cmath.acos(x)	Returns the arc cosine value of x
cmath.acosh(x)	Returns the hyperbolic arc cosine of x
cmath.asin(x)	Returns the arc sine of x
cmath.asinh(x)	Returns the hyperbolic arc sine of x
cmath.atan(x)	Returns the arc tangent value of x
cmath.atanh(x)	Returns the hyperbolic arctangent value of x
cmath.cos(x)	Returns the cosine of x
cmath.cosh(x)	Returns the hyperbolic cosine of x

<code>cmath.exp(x)</code>	Returns the value of E^x , where E is Euler's number (approximately 2.718281...), and x is the number passed to it
<code>cmath.isclose()</code>	Checks whether two values are close, or not
<code>cmath.isfinite(x)</code>	Checks whether x is a finite number
<code>cmath.isinf(x)</code>	Check whether x is a positive or negative infinity
<code>cmath.isnan(x)</code>	Checks whether x is NaN (not a number)
<code>cmath.log(x[, base])</code>	Returns the logarithm of x to the base
<code>cmath.log10(x)</code>	Returns the base-10 logarithm of x
<code>cmath.phase()</code>	Return the phase of a complex number
<code>cmath.polar()</code>	Convert a complex number to polar coordinates
<code>cmath.rect()</code>	Convert polar coordinates to rectangular form
<code>cmath.sin(x)</code>	Returns the sine of x
<code>cmath.sinh(x)</code>	Returns the hyperbolic sine of x
<code>cmath.sqrt(x)</code>	Returns the square root of x
<code>cmath.tan(x)</code>	Returns the tangent of x
<code>cmath.tanh(x)</code>	Returns the hyperbolic tangent of x

cMath Constants

Constant	Description
<code>cmath.e</code>	Returns Euler's number (2.7182...)
<code>cmath.inf</code>	Returns a floating-point positive infinity value
<code>cmath.infj</code>	Returns a complex infinity value
<code>cmath.nan</code>	Returns floating-point NaN (Not a Number) value
<code>cmath.nanj</code>	Returns complex NaN (Not a Number) value
<code>cmath.pi</code>	Returns PI (3.1415...)

<code>cmath.tau</code>	Returns tau (6.2831...)
------------------------	-------------------------