Python Tutorials Notes - Simple Version

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.

```python
# Create Comment
# Comments starts with a `#`, and Python will ignore them

# this is a comment
print("Hello, World!") # this is a comment

# print("this is a comment")

# Multi Line Comments
# This is a comment
# written in
# more than just one line
print("Hello, World!")

"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")

'''
This is a comment
written in
more than just one line
'''
print("Hello, World!")

```

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

```python
# Create Variable
x = 5
y = "John"
print(x)
print(y)

x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)

x = "John"
x = 'John' # is the same as

x, y, z = "Apple", "Banana", "Cherry"
x = y = z = "Orange"

# Output Variables

x = "awesome"
print("Python is " + x)

# Join Variables
x = "Python is "
y = "awesome"
z =  x + y
print(z)

x = 5
y = 10
print(x + y)

x = 5
y = "John"
```

```python
print(x + y) # Python will give you an error.

```

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)

```python
# Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

# Illegal variable names:
2myvar = "John"
my-var = "John"
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.

```python
# Create a variable outside of a function, and use it inside the function.

x = "awesome"

def my_func():
  print("Python is " + x)
```

```python

my_func()

# Local Variable
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)

# Global Keyword
# To create a global variable inside a function, you can use the global keyword.
def my_func():
  global x
  x = "fantastic"

my_func()

print("Python is " + x)

# Change global variable
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)

```

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.

```python

def my_func():
  x = 300
  print(x)

my_func() # A variable created inside a function is available inside that function.

# Function Inside Function
def my_func():
  x = 300
  def my_innerfunc():
    print(x)
  my_innerfunc()

my_func()

```

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.

```python
# A variable created outside of a function is global and can be used by anyone.
def myfunc():
  print(x)

myfunc()

print(x)

# Naming Variables
x = 300
```

```python

def my_func():
  x = 200
  print(x)

my_func()

print(x)

# Global Keyword
# If you need to create a global variable, but are stuck
 in the local scope, you can use the global keyword.
# The global keyword makes the variable global.

def my_func():
  global x
  x = 300

my_func()

print(x)

# To change the value of a global variable inside a function, refer to the variable by using the global keyword

x = 300

def my_func():
  global x
  x = 200

my_func()

print(x)

```

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`

```
x = 5
print(type(x)) # get the data type of any object by usin
g 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

```
# Integer
x = int(1)    # 1
y = int(2.8)  # 2
z = int("3")  # 3

# Float
x = float(1)      # 1.0
y = float(2.8)    # 2.8
z = float("3")    # 3.0
w = float("4.2")  # 4.2

# Strings
x = str("s1")     # 's1'
y = str(2)        # '2'
z = str(3.0)      # '3.0'
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

```
# Create Number
x = 1     # int
y = 2.8   # float
z = 1j    # complex

# Get the type of any Object
type(x)
```

```python

# Integer
x = 1
y = 35673444324353
z = -3232421
# Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

# Float
x = 1.10
y = 1.0
z = -35.59    # Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
x = 35e3
y = 13E4
z = -87.7e100 # Float can also be scientific numbers with an "e" to indicate the power of 10.


# Complex
x = 3+5j
y = 5j
z = -5j # Complex numbers are written with a "j" as the imaginary part.

# Type Conversion
a = float(x)
b = int (y)
c = complex(x) # You can convert from one type to another with the int(), float(), and complex() methods.
**Note: You cannot convert complex numbers into another number type.**

# Random Number
# 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.

import random
```

```python
print(random.randrange(1, 10)) # Import the random module, and display a random number between 1 and 9.

```

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”.

```python
# Print String
print("Hello")
print('Hello')
a = 'Hello'
b = "Hello"
print(a, b)

a = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit,

sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
"""   # three double quptes

a = '''
Lorem ipsum dolor sit amet,consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
'''    # three single quotes

```

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.

```python
# Index
a = "Hello, World!"
x = a[1]

# Slicing
a[2:5]
a[-5:-2]

# length of String
len(a)

# Check String
if 'he' in a:
    print(True)

if 'oo' not in a:
    print(False)

# String Join
a = "Hello"
b = "World"
c = a + b
c = a + " " + b

# String methods
a.strip()
a.lower()
a.upper()
a.replace("H", "J")
a.split(",")


```

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:

```python
# String Format
txt = "The price is {} dollars"
print(txt.format(price))  # Use the format() method to insert numbers into strings

txt = "The price is {:.2f} dollars"


# Multiple Values
quantity = 3
item = 567
price = 49.95
order = "I want {} pieces of item {} for {:.2f} dollars."
print(order.format(quantity, item, price))

# Index Numbers
quantity = 3
item = 567
price = 49.95
order = "I want to pay {2:.2f} dollars for {0} pieces of item {1}."
print(order.format(quantity, item, price))

name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name)) # refer to the same value more than once, use the index number

# Named Indexes
order = "I have a {car_name}, it is a {model}."
print(order.format(car_name = "Ford", model = "Mustang"))

```

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:

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

Other escape characters used in Python:

Code	Result
\'	Single Quote
\	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

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

```python

print(10 > 9)
print(10 == 9)
print(10 < 9)

# Conditions
a = 200
b = 33

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")

# Evaluate Values and Variables
bool("Hello")
bool(15)
x = "Hello"
y = 5
bool(x)
bool(y)  # The bool() function allows you to evaluate any value, and give you True or False in return.

#  Most Values are True
# Almost any value is evaluated to True if it has some sort of content.
# Any string is True, except empty strings.
# Any number is True, except 0.
# Any list, tuple, set, and dictionary are True, except empty ones.

bool("abc")
bool(123)
bool(["apple", "cherry", "banana"]) # all will return True
```

```

# Some Values are False
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.

bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({}) # all will return False

# 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.

class my_class():
  def __len__(self):
    return 0

my_obj = my_class()
print(bool(my_obj))

# Functions can Return a Boolean
def myFunction() :
  return True

print(myFunction())


def myFunction() :
  return True

if myFunction():
  print("YES!")
else:
  print("NO!")
```

```python

# built-in functions
x = 200
print(isinstance(x, int))

```

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

```

# Create List
list = ['apple', 'banana', 'cherry']
list = list(('apple', 'banana', 'cherry'))
a = 'is'
list = ['apple','boy', a, 2]
list = [[2,3,4], 2, 3, [22, 3,3]]


# Copy List
mylist = list.copy() # make a copy of a list with the copy() method:

mylist = list(list) # make a copy of a list with the list() method:

mylist = list * 2

list   = [1, 2, 3, 4]
mylist = list ** 2

# Check List
if list > 4:
    print(True)

# Join List
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

list3 = list1 + list2
```

```

# appending one by one
for x in list2:
    list1.append(x)

list.extend(list2)

# Loop List
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

length  = len(list) # print the number of items in this list:

for x in list:
    print(x)

for i, x in enumerate(list):
    print(i, x)

for i in range(len(list)):
    print(i, list[i])

# Built-in List

list.index('apple') # Get the index of an item
list.count('apple') # Count an item
list.reverse()      # Reverse the list
list.sort()         # Sort the list

```

2. Items Operate

```
# Check if 'apple' is present in the list
if 'apple' in list:
    print(True)

# Add Item
list.append('orange')
list.insert(1, 'orange')

```

```python
# Remove Item
list[2:5] = []
list[:] = []
list.remove('orange')
list.pop()
del list[0]
del list  # delete the list completely
list.clear()

# Access Item
list[1] # the first item has index 0, using index number
list[-1] # using negative indexing
list[2:5]  # using range of indexes, index 2(included) and end at index 5 (not included)
list[:4]
list[2:]
list[-4:-1]

list[1][0]
list[1][:2] # subset lists of lists

# Change Item
list[1] = 'orange' # replace the value

```

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

```python
# Create Tuple
tuple = ('apple', 'banana', 'cherry')

tuple = ("apple", ) # One item tuple, remember the comma

my_tuple = tuple(("apple", "banana", "cherry"))

# Loop Tuple
for x in tuple:
    print(x)

# Length of Tuple
len(tuple)

# Join Tuple

tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2

```

2. Items Operate

```python
# Access Item
tuple[1]

tuple[-1]

tuple[2:5]

tuple[-4:-1]

# Change Item
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

# Check Item
if 'apple' in tuple:
    print("Yes, 'apple' is in the fruits tuple.")

# Add Item
**Once a tuple is created, you cannot add items to it. Tuples are unchangeable.**

# Remove Item
**Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely**

del tuple  # `del` keyword can delete the tuple completely

# Built-in Methods
count('apple')   # Returns the number of times a specified value occurs in a tuple
index('apple')   # Searches the tuple for a specified value 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
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

1. Set Operate

```
# Create Set
set = {"apple", "banana", "cherry"}

my_set = set(('apple', 'banana', 'cherry'))

# Note: Sets are unordered, so you cannot be sure in which order the items will appear.

# Loop Set
for x in set:
```

```python
    print(x)

# Length of Set
len(set)

# Join Set
set1 = {'a', 'b', 'c'}
set2 = {1, 2, 3}
set3 = set1.union(set2) # The `union()` method returns a new set with all items from both sets.

set1.update(set2) # The `update()` method inserts the items in set2 into set1.
# Note: Both union() and update() will exclude any duplicate items.



```

2. Item Operate

```python

# Check Item
if 'apple' in set:
    print(True)

# Change Item
**Note: Sets are unordered, so you cannot be sure in which order the items will appear.**

# Add Item
set.add("orange")

set.update(['orange', 'mango, 'grapes'])

# Remove Item
set.remove('apple')
# Note: If the item to remove does not exist, remove() will raise an error.

set.discard('apple')
```

```
# Note: If the item to remove does not exist, discard()
 will NOT raise an error.

x = set.pop()
# Note: Sets are unordered, so when using the pop() meth
od, you will not know which item that gets removed.

set.clear() # empties the set

del set # 'del' keyword will delete the set completely

```

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

```python
# Create Dict
dict  = {
        "brand" : "Ford",
        "model" : "Mustang"
        "year"  : 1984
      }

# Loop Dict
for key in dict:
    print(key)  # Print all key names in the dictionary, one by one

for key in dict:
    print(dict[key]) # Print all values in the dictionary, one by one

for val in dict.values():
    print(val)  # You can also use the values() method to return values of a dictionary

for key, val in dict.items():
    print(key, val) # Loop through both keys and values, by using the items() method

# Length of Dict
len(dict)


# Copy Dict
dict2 = dict1  # You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

my_dict = dict.copy() # Make a copy of a dictionary with the copy() method

my_dict = dict(dict)

my_dict = dict(brand="Ford", model="Mustang", year=1964)

# Nested Dict

my_dict = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

my_dict = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}

```

2. Item Operate

```python
# Access Item
x = dict["model"]

x = dict.get("model") # There is also a method called get() that will give you the same result.

# Change Item
dict["year"] = 2020

# Check if Key Exists
if 'model' in dict:
    print("Yes, 'model' is one of the keys in the dict dictionary")

# Add Item

dict['color'] = 'red'

# Remove Item

dict.pop('model') # The pop() method removes the item with the specified key name

dict.popitem() # The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead)

del dict['model'] # The del keyword removes the item with the specified key name

del dict # The del keyword can also delete the dictionary completely

dict.clear() # The del keyword can also delete the dictionary completely

```

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.

```
# Create Array
cars = ["Ford", "Volvo", "BMW"]

# Access Element
cars[0]

# Modify Element
cars[0] = "Toyota"

# Length of An Array
x = len(cars)  # Note: The length of an array is always one more than the highest array index.

# Loop Array
for x in cars:
    print(x)

# Add Element
cars.append("Honda")

# Remove Array
cars.pop(1)

cars.remove("Volvo") # Note: The list's remove() method only removes the first occurrence of the specified value.

```

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

```python
# Return an iterator from a tuple, and print each value
my_tuple = ("apple", "banana", "cherry")
my_it = iter(my_tuple)

```

```python
print(next(my_it))
print(next(my_it))
print(next(my_it))

# Strings are also iterable objects, containing a sequence of characters
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))

# Looping Through an Iterator
# Iterate the values of a tuple
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)

# Iterate the characters of a string
mystr = "banana"

for x in mystr:
  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.

```python
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))

```

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.

```python
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  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 != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

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

```
# If Statement

a = 33
b = 200
if b > a:
    print("b is greater than a ")

if a > b: print("a is greater than b") # short Hand if

# Indentation
a = 33
```

```python
b = 200
if b > a:
print("b is greater than a")  # you will get an error

# Elif
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")

# Else
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")


print("A") if a > b else print("B")  # Short Hand If ... Else

# This technique is known as Ternary Operators(三元运算符), or Conditional Expressions.

print("A") if a > b else print("=") if a == b else print("B")

# And
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")

# Or
a = 200
```

```python
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")

# Nested If

x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")

# The pass Statement
a = 33
b = 200
if b > a:
    pass
# 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.

```

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.

```python
i = 1
while i < 6:
    print(i)
    i +=1  # Note: remember to increment i, or else the loop will continue forever.
```

```python

# Break Statement
i = 1
while i < 6:
  print(i)
  if i == 3:
    break  # the break statement we can stop the loop even if the while condition is true
  i += 1

# Continue Statement
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue # `continue` statement we can stop the current iteration, and continue with next.
  print(i)

# Else Statement
i = 1
while i < 6:
  print(i)
  i += 1
else:      # With the else statement we can run a block of code once when the condition no longer is true.
  print("i is no longer less than 6")

```

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.

```python
# For - List
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)

# For - String
for x in "banana":
  print(x)

# Break Statement
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)

# Continue Statement

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)


# For - Range
# To loop through a set of code a specified number of times, we can use the range() function,
# The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

for x in range(6):
    print(x)

```

```python
# Note that range(6) is not the values of 0 to 6, but the values 0 to 5.
for x in range(2, 6):
    print(x)

# The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3)
for x in range(2, 30, 3):
    print(x)

# Else in For Loop
for x in range(6):
    print(x)
else:
print("Finally finished!")


# Nested Loops
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)

# Pass Statement
for x in [0, 1, 2]:
    pass

```

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.

```python
# Creating a Function
def my_func():       # In Python a function is defined using the def keyword
    print("Hello")

# Calling a Function
def my_function():
  print("Hello from a function")

my_function()

# Arguments
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")

# Arguments are often shortened to args in Python documentations.

# Parameters or Arguments

# Number of Arguments
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")

def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")    # will get an error

# Arbitrary Arguments, *args
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")

```

```python
# Keyword Arguments
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

# Arbitrary Keyword Arguments, **kwargs
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")

# Default Parameter Value
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")

# Passing a List as an Argument
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)

# Return Values
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))

# The pass Statement
def myfunction():
  pass

# Recursion 递归
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)

```

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:

```python

x = lambda a: a + 10
print(x(5))

x = lambda a, b : a * b
print(x(5, 6))

x = lambda a, b, c : a + b + c
print(x(5, 6, 2))

```

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.

```
def my_func(n):
    return lambda a : a *n

my_doubler = my_func(2)

print(my_doubler(11))


def my_func(n):
  return lambda a : a * n

my_doubler = my_func(2)
my_tripler = my_func(3)

print(my_doubler(11))
print(my_tripler(11))

```

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.

```
# Create a Class
class MyClass:
    x = 5

# Create Object
p1 = MyClass()
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.

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

p1 = Person("John", 36)
print(p1.name)
print(p1.age)

**Note: The __init__() function is called automatically every time the class is being used to create a new object.**

# ObJect Methods
# Objects can also contain methods. Methods in objects are functions that belong to the object.

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def my_func(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.my_func()
```

```

**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.**

# The self Parameter
  # Use the words mysillyobject and abc instead of self
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def my_func(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.my_func()

# Modify Object Properties
p1.age = 40

# Delete Object Properties
del p1.age

# Delete Objects
del p1

# The pass Statement
class Person:
    pass

```

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.

```python
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def print_name(self):
    print(self.firstname, self.lastname)

# Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.print_name()

# Create a Child Class
class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.print_name()

# Add the __init__() Function
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.

# Note: The child's __init__() function overrides the inheritance of the parent's __init__() function.

class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
# To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function

# Use the super() Function
```

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)

# Add Properties
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)


# Add Methods
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

# If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

```

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.

```
# Create a module
```

```python
def greeting(name):    # Save this code in a file named mymodule.py
    print("Hello, " + name)

# Use a Module
import mymodule

mymoudle.greeting("Jonathan")
# Note: When using a function from a module, use the syntax: module_name.function_name.

# Variables in Module
# The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc).

person1 = {  # Save this code in the file mymodule.py
  "name": "John",
  "age": 36,
  "country": "Norway"
}

import mymodule

a = mymodule.person1["age"]
print(a)

# Naming a Module
# You can name the module file whatever you like, but it must have the file extension .py

# Re-naming a Module
import mymodule as mx
a  = mx.person1("age")
print(a)

# Built-in Modules
import platform
x = platform.system()
print(x)

```

```
# Using the dir() Function
# There is a built-in function to list all the function
 names (or variable names) in a module. The dir() functi
on.

import platform
x = dir(platform)
print(x)

# Note: The dir() function can be used on all modules, a
lso the ones you create yourself.

# Import From Module
You can choose to import only parts from a module, by us
ing the from keyword.

def greeting(name):
    print("Hello, " + name)

person1 = {
 "name" : "John",
 "age"  : 36,
 "country" : "Norway"
}

from mymodule import person1

print (person1["age"])

```

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.

```python

# try block
try:
    print(x)
except:
    print("An Exception occurred")

# Many Exceptions
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")

# Else
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")

# Finally
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")

try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()

# Raise an exception
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")

# Raise a TypeError if x is not an integer
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")

```

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.

```python
import datetime
x = datetime.datetime.now()
print(x)  # 2020-10-16 11:54:42.839588

print(x.year)
print(x.strftime("%A"))

# Creating Date Objects
x = datetime.datetime(2020, 5, 17)
print(x)

# The strftime() Method
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

```

```

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.

```python
x = min(5, 10, 25)
y = max(5, 10, 25)
x = abs(-7.25)
x = pow(4, 3)

# math module
import math
x = math.sqrt(64)
x = math.ceil(1.4)
y = math.floor(1.4)
x = math.pi

```

Python JSON

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

```python
import json

# Parse JSON - Convert from JSON to Python

x =  '{ "name":"John", "age":30, "city":"New York"}' # some JSON:
```

```
y = json.loads(x) # parse x:
print(y["age"]) # the result is a Python dictionary:

# Convert from Python to JSON
# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)

# You can convert Python objects of the following types,
  into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))

```

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

```
x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

print(json.dumps(x))

# Format the Result
json.dumps(x, indent=4)

json.dumps(x, indent=4, separators=(". ", " = "))

```

```python
# Order the Result
json.dumps(x, indent=4, sort_keys=True)

```

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.

```python
# RegEx Module
import re

# RegEx in Python
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)

```

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

```python
import re

# findall()
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)

txt = "The rain in Spain"
```

```python
x = re.findall("Portugal", txt)
print(x) # Return an empty list if no match was found

# search()
txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)

# split()
txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)

# sub()
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)


```

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.

```python
txt = "The rain in Spain"
```

```
x = re.search("ai", txt)
print(x) #this will print an object

The Match object has properties and methods used to retr
ieve information about the search, and the result:

# .span() returns a tuple containing the start-, and end
 positions of the match.
# .string returns the string passed into the function
# .group() returns the part of the string where there wa
s a match

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())  # (12, 17)

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())

Note: If there is no match, the value None will be retur
ned, 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.

```
# Check if PIP is Installed
>>> pip --version

```

```
# Install PIP

# Download a Package
>>> pip install camelcase

# Using a package
import camelcase
c = camelcase. Camelcase()
txt = "Hello, World"
print(c.hum(txt))

# Find Packages

# Remove Packages
>>> uninstall

>>> pip uninstall camelcase

# List packages

>>> pip list

Package         Version
-----------------------
camelcase       0.2
mysql-connector 2.1.6
pip             18.1
pymongo         3.6.1
setuptools      39.0.1

```

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:

```
# Python 3.6
username = input("Enter username: ")
print("Username is: " + username)

# Python 2.7
username = raw_input("Enter username:")
print("Username is: " + username)

```

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

Chapter 8: Python File Handing

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:

```
f = open("demofile.txt")
# The code above is the same as:

f = open("demofile.txt", "rt")
# 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

```
f = open("demofile.txt", "r")
print(f.read())

# Open a file on a different location
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())

```

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.

```python

# Return the 5 first characters of the file
f = open("demofile.txt", "r")
print(f.read(5))

# Read Lines
f = open("demofile.txt", "r")
print(f.readline())  # Read one line of the file

# By calling readline() two times, you can read the two first lines.
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())

# By looping through the lines of the file, you can read the whole file, line by line
f = open("demofile.txt", "r")
for x in f:
  print(x)

# Close Files

f = open("demofile.txt", "r")
print(f.readline())
f.close()

```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

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

```python

# appending
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())

# overwrite
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())

# Note: the "w" method will overwrite the entire file.

```

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

```python
# Create a file called "myfile.txt".
f = open("myfile.txt", "x")

# Create a new file if it does not exist
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

```python
import os
os.remove("demo_file.txt")

# Check if File exit
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")

# Delete Folder
os.rmdir("myfolder") # To delete an entire folder, use the os.rmdir() method.

```

Note: You can only remove empty folders.