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.

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

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

```python

# 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"
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)

```
# 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.

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

x = "awesome"

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

my_func()

# Local Variable
x = "awesome"

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

myfunc()

print("Python is " + x)

# Global Keyword
```

```python
# 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.

```
# 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

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():
```

```python
  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`

```python
x = 5
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

```
# 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)

# 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
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)
```

```python

# 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"
```

```python


# 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:

```python
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

```

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!")

# 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

```
```

```python
# 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"]

```

```python
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

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

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

# 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

```
# 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

```
# 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

```python
# 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:
    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

```

# 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() method, 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

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

```

```python
# 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

```
# 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

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

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.

```
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.

```
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

Python While Loops

Python For Loops

Chapter 6: Python Object-oriented

Python Functions

Python Lambda

Python Classes/Objects

Python Inheritance

Python Modules

Chapter 7: Python Handle

Python Try Except

Python Dates

Python Math

Python RegEx

Python JSON

Python PIP

Python Input

Chapter 8: Python File Handing

Python Read Files

Python Write/Create Files

Python Delete Files