Python Programming

Unit 2

STRINGS

Strings

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

`'hello'` is the same as `"hello"`.

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Try it Yourself »

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Try it Yourself »

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Try it Yourself »

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Try it Yourself »

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are 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.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Try it Yourself »

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

Try it Yourself »

Learn more about For Loops in our Python For Loops chapter.

String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Try it Yourself »

Check String

To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"
print("free" in txt)
```

Try it Yourself »

Use it in an `if` statement:

Example

Print only if "free" is present:

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Try it Yourself »

Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

Try it Yourself »

Use it in an `if` statement:

Example

print only if "expensive" is NOT present:

```
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
```

```
print(b[2:5])
```

Try it Yourself »

Note: The first character has index 0.

Slice From the Start

By leaving out the start index, the range will start at the first character:

Example

Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"
print(b[:5])
```

Try it Yourself »

Slice To the End

By leaving out the *end* index, the range will go to the end:

Example

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
print(b[2:])
```

Try it Yourself »

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example

Get the characters:
From: "o" in "World!" (position -5)
To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

Try it Yourself »

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

Upper Case

Example

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Try it Yourself »

Lower Case

Example

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Try it Yourself »

Remove Whitespace

Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

Example

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Try it Yourself »

Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Try it Yourself »

Split String

The `split()` method returns a list where the text between the specified separator becomes the list items.

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Try it Yourself »

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable `a` with variable `b` into variable `c`:

```
a = "Hello"
b = "World"
c = a + b
```

```
print(c)
```

Try it Yourself »

Example

To add a space between them, add a " ":

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

Try it Yourself »

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

Try it Yourself »

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Try it Yourself »

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

Try it Yourself »

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Try it Yourself »

Escape Characters

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

String Methods

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

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
find()	Searches the string for a specified value and returns the position of where it was found
format()	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
replace()	Returns a string where a specified value is replaced with a specified value
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
upper()	Converts a string into upper case

LIST

Lists are used to store multiple items in a single variable.

Lists are created using square brackets:

Example
Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

List items are ordered, changeable, and allow duplicate values.
List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered
When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
If you add new items to a list, the new items will be placed at the end of the list.

Changeable
The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates
Since lists are indexed, lists can have items with the same value:

Access Items

List items are indexed and you can access them by referring to the index number:

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Try it Yourself »

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end
-1 refers to the last item, -2 refers to the second last item etc.

Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Try it Yourself »

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Try it Yourself »

Note: The search will start at index 2 (included) and end at index 5 (not included).
Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT including, "kiwi":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

Try it Yourself »

By leaving out the end value, the range will go on to the end of the list:

Example

This example returns the items from "cherry" to the end:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

Try it Yourself »

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list:

Example

This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

Try it Yourself »

Check if Item Exists

To determine if a specified item is present in a list use the `in` keyword:

Example

Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Try it Yourself »

Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Try it Yourself »

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Example

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Try it Yourself »

If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second value by replacing it with *two* new values:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

Try it Yourself »

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second and third value by replacing it with *one* value:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

Try it Yourself »

Insert Items

To insert a new list item, without replacing any of the existing values, we can use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert "watermelon" as the third item:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

Try it Yourself »

Append Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Try it Yourself »

Insert Items

To insert a list item at a specified index, use the `insert()` method.
The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Try it Yourself »

Note: As a result of the examples above, the lists will now contain 4 items.

Extend List

To append elements from *another list* to the current list, use the `extend()` method.

Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

Try it Yourself »

Remove Specified Item

The `remove()` method removes the specified item.

Example

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Try it Yourself »

Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

Try it Yourself »

If you do not specify the index, the `pop()` method removes the last item.

Example

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Try it Yourself »

The `del` keyword also removes the specified index:

Example

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

Try it Yourself »

The `del` keyword can also delete the list completely.

Example

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Try it Yourself »

Clear the List

The `clear()` method empties the list.
The list still remains, but it has no content.

Example

Clear the list content:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Try it Yourself »

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.
Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Try it Yourself »

The iterable created in the example above is `[0, 1, 2]`.

Using a While Loop

You can loop through the list items by using a `while` loop.
Use the `len()` function to determine the length of the list, then start at 0 and loop your way through the list items by refering to their indexes.
Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Try it Yourself »

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand `for` loop that will print all items in a list:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

Try it Yourself »

Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Try it Yourself »

Example

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

Try it Yourself »

Sort Descending

To sort descending, use the keyword argument `reverse = True`:

Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Try it Yourself »

Example

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

Try it Yourself »

Customize Sort Function

You can also customize your own function by using the keyword argument `key = function`.

The function will return a number that will be used to sort the list (the lowest number first):

Example

Sort the list based on how close the number is to 50:

```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

Try it Yourself »

Case Insensitive Sort

By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Example

Case sensitive sorting can give an unexpected result:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

Try it Yourself »

Luckily we can use built-in functions as key functions when sorting a list.
So if you want a case-insensitive sort function, use str.lower as a key function:

Example

Perform a case-insensitive sort of the list:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

Try it Yourself »

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?
The `reverse()` method reverses the current sorting order of the elements.

Example

Reverse the order of the list items:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Try it Yourself »

Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.
There are ways to make a copy, one way is to use the built-in List method `copy()`.

Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Try it Yourself »

Another way to make a copy is to use the built-in method `list()`.

Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
```

```
mylist = list(thislist)
print(mylist)
```

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

Example

Join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

Try it Yourself »

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Example

Append list2 into list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

Try it Yourself »

Or you can use the extend() method, which purpose is to add elements from one list to another list:

Example

Use the extend() method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

Try it Yourself »

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

Tuple

Tuples are used to store multiple items in a single variable.

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.
Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example
Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

Try it Yourself »

Tuple Length

To determine how many items a tuple has, use the `len()` function:

Example
Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Try it Yourself »

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example
One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
```

```
print(type(thistuple))
```

Try it Yourself »

Tuple Items - Data Types

Tuple items can be of any data type:

Example

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

Try it Yourself »

A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

Try it Yourself »

---

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

Example

What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

Try it Yourself »

The tuple() Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Try it Yourself »

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end.

`-1` refers to the last item, `-2` refers to the second last item etc.

Example

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Try it Yourself »

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Example

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Try it Yourself »

Note: The search will start at index 2 (included) and end at index 5 (not included).
Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT included, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```

Try it Yourself »

By leaving out the end value, the range will go on to the end of the list:

Example

This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```

Try it Yourself »

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
```

```
print(thistuple[-4:-1])
```

Try it Yourself »

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Try it Yourself »

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

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

print(x)
```

Try it Yourself »

Add Items

Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Try it Yourself »

2. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Try it Yourself »

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Remove Items

Note: You cannot remove items in a tuple.

Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Try it Yourself »

Or you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Try it Yourself »

Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Example

Packing a tuple:

```
fruits = ("apple", "banana", "cherry")
```

Try it Yourself »

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Example

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

Try it Yourself »

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Try it Yourself »

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.
Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Try it Yourself »

Example

Print all items, using a `while` loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
```

```
  print(thistuple[i])
  i = i + 1
```

Join Two Tuples

To join two or more tuples you can use the + operator:

Example

Join two tuples:

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

tuple3 = tuple1 + tuple2
print(tuple3)
```

Try it Yourself »

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

Try it Yourself »

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

DICTIONARY

Dictionary

Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
Dictionary items are ordered, changeable, and does not allow duplicates.
Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Try it Yourself »

Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Try it Yourself »

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

Try it Yourself »

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict':

```
<class 'dict'>
```

Example

Print the data type of a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

Try it Yourself »

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

Try it Yourself »

There is also a method called `get()` that will give you the same result:

Example

Get the value of the "model" key:

```
x = thisdict.get("model")
```

Try it Yourself »

Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

```
x = thisdict.keys()
```

Try it Yourself »

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.keys()
print(x) #before the change
car["color"] = "white"
print(x) #after the change
```

Try it Yourself »

Get Values

The `values()` method will return a list of all the values in the dictionary.

Example

Get a list of the values:

```
x = thisdict.values()
```

Try it Yourself »

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example

Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
```

```
}
x = car.values()

print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Try it Yourself »

Example

Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

Try it Yourself »

Get Items

The `items()` method will return each item in a dictionary, as tuples in a list.

Example

Get a list of the key:value pairs

```
x = thisdict.items()
```

Try it Yourself »

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example

Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Try it Yourself »

Example

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

Try it Yourself »

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Try it Yourself »

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Try it Yourself »

Update Dictionary

The `update()` method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs. If the item does not exist, the item will be added. The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Adding an item to the dictionary

This is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Try it Yourself »

Example

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Try it Yourself »

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Try it Yourself »

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Try it Yourself »

Example

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Try it Yourself »

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Try it Yourself »

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

Try it Yourself »

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Try it Yourself »

Example

You can use the `keys()` method to return the keys of a dictionary:

```
for x in thisdict.keys():
  print(x)
```

Try it Yourself »

Example

Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():
  print(x, y)
```

Try it Yourself »

Copy a Dictionary

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`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Try it Yourself »

Another way to make a copy is to use the built-in function `dict()`.

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
```

```
}
mydict = dict(thisdict)
print(mydict)
```

Try it Yourself »

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Example

Create a dictionary that contain three dictionaries:

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

Try it Yourself »

Or, if you want to add three dictionaries into a new dictionary:

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
```

```
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Try it Yourself »

FILE HANDLING

FILE MODES

- `r` for reading – The file pointer is placed at the beginning of the file. This is the default mode.
- `r+` Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
- `w` Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, create a new file for writing.
- `w+` Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
- `"t" - Text - Default value. Text mode`
- `"rt" - Default mode - read in text mode`
- `rb` Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
- `rb+` Opens a file for both reading and writing in binary format.
- `wb+` Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
- `a` Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- `ab` Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- `a+` Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
- `ab+` Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
- `x` open for exclusive creation, failing if the file already exists (Python 3)

File opening in Python

open() function is used to open a file in Python. It's mainly required two arguments, first the file name and then file opening mode.

Syntax:

```
file_object = open(filename [,mode] [,buffering])
```

In the above syntax, the parameters used are:

filename: It is the name of the file.
mode: It tells the program in which mode the file has to be open.
buffering: Here, if the value is set to zero (0), no buffering will occur while accessing a file; if the value is set to top one (1), line buffering will be performed while accessing a file.

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.

Reading a File

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

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

The following code will return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

You can return one line by using the readline() method:

Example

Read one line of the file:

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

By calling `readline()` two times, you can read the two first lines:

Example

Read two lines of the file:

```
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:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

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

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

Example

Open the file "demofile2.txt" and append content to the file:

```
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())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
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.

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

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

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

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
```

```
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

Get the current working directory

The directory that the Python program is in is known as the current working directory for that program. In order to get the current working directory you'll need to use the os module with the function getcwd() as follows:

```
import os
cwd = os.getcwd()
print(cwd)
```

To change the current working directory you can use function chdir(). You just pass the current working directory you want to change to as follows:

```
import os
os.chdir('c:\\temp')
cwd = os.getcwd()
print(cwd)
```

Join and split a path

To make a program work across platforms including Windows, Linux, and Unix, you need to use a platform-independent file and directory path. Python provides a submodule os.path that contains several useful functions and constants to join and split paths.

The join() function joins path components together and returns a path with exact platform separator for instance in windows backslash () and Unix (/)

The split() function splits path into path components without path separator. Here's an example of using join() and split() functions:

```
import os
fp = os.path.join('temp','python')
print(fp) # temp\python (in Windows)
```

```
pc = os.path.split(fp)
print(pc) # ('temp', 'python')
```

Test if a path is a directory

In order to check a path exists in a system and a path is a directory you can use the functions exists() and isdir() in the submodule os.path as below:

```
import os

path = os.path.join("C:\\","temp")
if os.path.exists(path):
    print(path + ' : exists')
    if os.path.isdir(path):
        print(path + ' : is a directory')
```

Create a directory

To create a new directory you use mkdir() or makedirs() functions of os module. And you should always check if a directory exists first before creating a new directory.
The following example creates a new directory called python under the c:\temp directory.

```
import os

dir = os.path.join("C:\\","temp","python")
if not os.path.exists(dir):
    os.mkdir(dir)
```

Rename a directory

To rename the directory you use os.rename() function. You need to pass the directory you want to change and the new path.

```
import os

oldpath = os.path.join("C:\\","temp","python")
newpath = os.path.join("C:\\","temp","python3")

if os.path.exists(oldpath):
    os.rename(oldpath, newpath)
    print("'{0}' is renamed to '{1}'".format(oldpath,newpath))
    #'C:\temp\python' is renamed to 'C:\temp\python3'
```

Delete a directory

To delete a directory, you use function rmdir() function as follows:

```
import os
```

```
dir = os.path.join("C:\\","temp","python")
if os.path.exists(dir):
    os.rmdir(dir)
    print(dir + ' is removed.')
```

Traverse a directory recursively

Python provides os.walk() function that allows you to traverse a directory recursively. The os.walk() function returns the root directory, the sub-directories, and files.
The following example shows how to print all files and directories in the c:\temp directory:

```
import os

rootdir = "c:\\temp"
for root, dirs, files in os.walk(rootdir):
    print("{0} has {1} files".format(root, len(files)))
```

Exception Handling

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.

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:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.
Without the try block, the program will crash and raise an error:

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a NameError and another for other errors:

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

Try it Yourself »

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the try block does not generate any error:

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

Try it Yourself »

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example

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

Try it Yourself »

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("Pledge.txt",'w')
  f.write("A sample paragraph 090821")
except:
  print("Something went wrong when writing to the file")
else:
  print("There is no exception. Program execution is successful")
finally:
  try:
```

```
  f.close()
  print("Connection to file is now closed")
except: print("File object F does not exist")
```

The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, use the `raise` keyword.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"

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

Few Built-in Exceptions

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indendation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
NameError	Raised when a variable does not exist
SystemExit	Raised when the sys.exit() function is called

ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero