Python

Syntax

Python syntax can be executed by writing directly in the Command Line:

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

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

```
$ python myfile.py
```

Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

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

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

Variables

In Python, variables are created when you assign a value to it:

```
x = 5
y = "Hello, World!"
```

- Python has no command for declaring a variable.

Comments

Python has commenting capability for the purpose of in-code documentation. Comments can be used to:

- Explain Python code
- Make the code more readable
- Prevent execution when testing code

Creating a Comment

Comments start with a `#` and Python will render the rest of the line as a comment and Python will ignore them:

```
# This is a comment
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

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

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

```
# print("Hello, World!")
print("Cheers, Mate!")
```

Multiline Comments

Python does not really have a syntax for multiline comments.

To add a multiline comment you could insert a `#` for each line:

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

Or, not quite as intended, you can use a multiline string.

> Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it

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

- As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Variables

Creating Variables

Python has no command for declaring a variable.

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

```python
x = 5
y = "John"
```

Variables do not need to be declared with any particular *type* and can even change type after they have been set.

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

Casting

If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Get the Type

You can get the data type of a variable with the `type()` function.

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

Single or Double Quotes?

String variables can be declared either by using single or double quotes:

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

Case-Sensitive

Variable names are case-sensitive.

```
a = 4
A = "Sally"
# A will not overwrite a
```

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:

- Must start with a letter or the underscore character
- Cannot start with a number
- 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)
- Cannot be any of the Python keywords.

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Remember that variable names are case-sensitive

Multi Words Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

- Camel Case: `myVariableName`
- Pascal Case: `MyVariableName`
- Snake Case: `my_variable_name`

Assign Multiple Values

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

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

Note: Make sure the number of variables matches the number of values, or else you will get an error.

One Value to Multiple Variables

And you can assign the same value to multiple variables in one line:

```
x = y = z = "Orange"
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
```

Output Variables

The Python `print()` function is often used to output variables.

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

In the `print()` function, you output multiple variables, separated by a comma:

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

You can also use the `+` operator to output multiple variables:

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Notice the space character after `"Python "` and `"is "` without them the result would be "Pythonisawesome".

For numbers, the `+` character works as a mathematical operator:

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

In the `print()` function, when you try to combine a string and a number with the `+` operator, Python will give you an error

The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

```
x = 5
y = "John"
print(x, y)
```

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.

```
x = "awesome"

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

myfunc()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```python
x = "awesome"

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

myfunc()

print("Python is " + x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the `global` keyword.

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

myfunc()

print("Python is " + x)
```

Also, use the `global` keyword if you want to change a global variable inside a function.

```python
x = "awesome"

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

myfunc()

print("Python is " + x)
```

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`
None Type	`NoneType`

Getting the Data Type

You can get the data type of any object by using the `type()` function:

```python
x = 5
print(type(x))
```

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)`	`bytearray`
`x = memoryview(bytes(5))`	`memoryview`
`x = None`	`NoneType`

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`

Example	Data Type
`x = bytes(5)`	`bytes`
`x = bytearray(5)`	`bytearray`
`x = memoryview(bytes(5))`	`memoryview`

Numbers

There are three numeric types in Python:

- `int`
- `float`
- `complex`

Variables of numeric types are created when you assign a value to them:

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

To verify the type of any object in Python, use the `type()` function:

```
print(type(x))
print(type(y))
print(type(z))
```

int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522
print(type(x))
print(type(y))
print(type(z))
```

float

Float, or “floating point number” is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
z = -35.59
print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an “e” to indicate the power of 10.

```
x = 35e3
y = 12E4
z = -87.7e100
```

complex

Complex numbers are written with a “j” as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
print(type(x))
print(type(y))
print(type(z))
```

Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
# convert from int to float:
a = float(x)
# convert from float to int:
b = int(y)
# convert from int to complex:
c = complex(x)
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

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))
```

Casting

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 removing all decimals), 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

int

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

float

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

string

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

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:

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

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:

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

Multiline Strings

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

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

Or three single quotes.

> **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.

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

Looping Through a String

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

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

String Length

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

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

Check String

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

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

Use it in an `if` statement:

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

Check if NOT

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

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

Use it in an `if` statement:

```
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.

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

Note: The first character has index 0.

Slice From the Start

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

Slice To the End

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

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

Negative Indexing

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

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

Modify Strings

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

Upper Case

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

Lower Case

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

Remove Whitespace

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

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

Replace String

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

Split String

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

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

String Concatenation

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

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

String Format

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

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

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

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

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

```
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))
```

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."
```

To fix this problem, use the escape character `\"`:

```
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

String Methods

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

Note: All string methods return 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

Method	Description
`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

Method	Description
`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

Boolean

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

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)
```

When you run a condition in an if statement, Python returns `True` or `False`:

```
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

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return.

```
print(bool("Hello"))
print(bool(15))
```

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

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"])
```

Some Values are False

In fact, there are not many values that evaluate 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({})
```

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 myclass():
    def __len__(self):
        return 0

myobj = myclass()
print(bool(myobj))
```

Functions can Return a Boolean

You can create functions that returns a Boolean Value:

```
def myFunction():
    return True

print(myFunction())
```

You can execute code based on the Boolean answer of a function:

```
def myFunction():
    return True

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

Python also has many built-in functions that return a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

```
x = 200
print(isinstance(x, int))
```

Operators

Operators are used to perform operations on variables and values.

In the example below, we use the `+` operator to add together two values:

```
print(10 + 5)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators

- Membership operators
- Bitwise operators

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`

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`

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`

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)`

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`

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

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`

Bitwise Operators

Operator	Name	Description	Example
`&`	AND	Sets each bit to 1 if both bits are 1	`x & y`
`|`	OR	Sets each bit to 1 if one of two bits is 1	`x | y`
`^`	XOR	Sets each bit to 1 if only one of two bits is 1	`x ^ y`
`~`	NOT	Inverts all the bits	`~x`
`<<`	**Zero fill left shift**	Shift left by pushing zeros in from the right and let the leftmost bits fall off	`x << 2`

Operator	Name	Description	Example
`>>`	**Signed right shift**	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	`x >> 2`

Operator Precedence

Operator precedence describes the order in which operations are performed.

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```

Multiplication `*` has higher precedence than addition `+` and therefor multiplications are evaluated before additions:

```
print(100 + 5 * 3)
```

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x -x ~x`	Unary plus, unary minus, and bitwise NOT
`* / // %`	Multiplication, division, floor division, and modulus
`+ -`	Addition and subtraction
`<< >>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`== != > >= < <= is is not in not in`	Comparisons, identity, and membership operators
`not`	Logical NOT

Operator	Description
`and`	AND
`or`	OR

- If two operators have the same precedence, the expression is evaluated from left to right.

```
print(5 + 4 - 7 + 3)
```

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, unchangeable[1], and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered[2] and changeable. No duplicate members.

1. Set items are unchangeable, but you can remove and/or add items whenever you like.
2. As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

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.

Lists

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

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are **Tuple**, **Set**, and **Dictionary**, all with different qualities and usage.

Lists are created using square brackets:

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

Basics

List items are:

- Ordered
- Changeable
- 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.

> **Note**: There are some list methods that will change the order, but in general: the order of the items will not change.

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:

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

List Length

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

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

List Items - Data Types

List items can be of any data type:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
```

type()

From Python's perspective, lists are defined as objects with the data type 'list':

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

```
<class 'list'>
```

The list() Constructor

It is also possible to use the `list()` constructor when creating a new list.

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

Access List Items

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

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

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.

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

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.

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

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:

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

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

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

Range of Negative Indexes

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

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

Check if Item Exists

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

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

Change List Items

Change Item Value

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

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

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:

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

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

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

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:

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

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:

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

Note: As a result of the example above, the list will now contain 4 items.

Add List Items

Append Items

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

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

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:

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

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.

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

The elements will be added to the *end* of the list.

Add Any Iterable

The `extend()` method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

Remove List Items

Remove Specified Item

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

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

Remove Specified Index

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

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

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

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

The `del` keyword also removes the specified index:

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

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

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

Clear the List

The `clear()` method empties the list.

The list still remains, but it has no content.

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

Loop Lists

Loop Through a List

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

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

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.

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

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 referring to their indexes.

Remember to increase the index by 1 after each iteration.

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

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

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

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter “a” in the name.

Without list comprehension you will have to write a `for` statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that valuate to `True`.

```
newlist = [x for x in fruits if x != "apple"]
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".

The *condition* is optional and can be omitted:

```
newlist = [x for x in fruits]
```

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

```
newlist = [x for x in range(10) if x < 5]
```

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like:

```
newlist = ['hello' for x in fruits]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

The *expression* in the example above says:

"*Return the item if it is not banana, if it is banana return orange*".

Sort Lists

Sort List Alphanumerically

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

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

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

Sort Descending

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

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

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

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):

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

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

Case Insensitive Sort

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

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

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:

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

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.

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

Copy Lists

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().`

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

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

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

Join Lists

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.

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

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

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

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

for x in list2:
    list1.append(x)

print(list1)
```

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

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

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

List Methods

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

Tuples

Basics

A tuple is a collection which is ordered and *unchangeable***.

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:

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

Tuple Length

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

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

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.

```
thistuple = ("apple",)
print(type(thistuple))
# NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

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

A tuple can contain different data types:

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

type()

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

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

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

The tuple() Constructor

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

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

Access Tuple Items

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

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

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.

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

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.

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

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:

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

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

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

Range of Negative Indexes

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

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

Check if Item Exists

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

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

Note: Accessing tuple items is just like accessing list items.

Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created. But there are some workarounds.

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.

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

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.

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

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:

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

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:

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

Or you can delete the tuple completely using `del` keyword:

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

Unpack Tuples

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

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

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

```python
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

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.

Using Asterisk*

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

```python
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Loop Tuples

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

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

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.

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

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 tuple, then start at 0 and loop your way through the tuple items by refering to their indexes.

Remember to increase the index by 1 after each iteration.

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

Join Tuples

Join Two Tuples

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

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

Multiply Tuples

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

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

Tuple Methods

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

Sets

A set is a collection which is unordered, unchangeable[1] and unindexed.

1. Set items are unchangeable, but you can remove items and add new items.

Basics

Sets are written with curly brackets.

```python
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

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

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

> **Note**: Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed

Sets cannot have two items with the same value.

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

Get the Length of a Set

To determine how many items a set has, use the `len()` function.

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

```python
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

```python
set1 = {"abc", 34, True, 40, "male"}
```

type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

```python
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

The set() Constructor

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

```python
# note the double round-brackets
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

Access Set Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

```python
for x in thisset:
print(x)
```

```python
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

Change Items

> **Note**: Once a set is created, you cannot change its items, but you can add new items.

Add Set Items

To add one item to a set use the `add()` method.

```python
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Add Sets

To add items from another set into the current set, use the `update()` method.

```python
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

Add Any Iterable

The object in the `update()` method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```python
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
```

Remove Set Items

To remove an item in a set, use the **`remove()`** or the **`discard()`** method.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Notw: If the item to remove does not exist, **`remove()`** will raise an error.

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

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

You can also use the **`pop()`** method to remove an item, but this method will remove the last item.

Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the **`pop()`** method is the removed item.

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Note: Sets are *unordered*, so when using the **`pop()`** method, you do not know which item that gets removed.

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

The **`clear()`** method empties the set:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

```
Traceback (most recent call last):
  File "/tmp/test.py", line 3, in <module>
    print(thisset)
          ^^^^^^^
NameError: name 'thisset' is not defined
```

Loop Sets

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

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
```

Join Sets

Join Two Sets

There are several ways to join two or more sets in Python.

You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another:

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

Note: Both `union()` and `update()` will exclude any duplicate items.

Keep ONLY the Duplicates

The `intersection_update()` method will keep only the items that are present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)
```

The `intersection()` method will return a *new* set, that only contains the items that are present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)
```

Keep All, But NOT the Duplicates

The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)
```

The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)
```

Set Methods

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

Method	Description
`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