



جامعة الانبار
كلية علوم الحاسوب وتكنولوجيا المعلومات
قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP
المرحلة الثانية
الفصل الدراسي الأول والثاني

مدرس المادة
م.د. سميه عبدالله حمد

Object Oriented Programming (OOP)

Python Fundamental Introduction

What is Python?

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

It is used for:

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

What can Python do?

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

Why Python?

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

Python Syntax compared to other programming languages

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

Object Oriented Programming (OOP)

Example

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

Python 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.

Example

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

Python will give you an error if you skip the indentation:

Example

Syntax Error:

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

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.

Example

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

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example

Syntax Error:

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

Object Oriented Programming (OOP)

Python Variables

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

Example

Variables in Python:

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

Python has no command for declaring a variable.

Comments

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

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

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

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

Example

```
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:

Example

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

Multi Line Comments

Python does not really have a syntax for multi line comments. To add a multiline comment you could insert a # for each line:

Object Oriented Programming (OOP)

Example

```
#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:

Example

```
"""  
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

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Example

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

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

Object Oriented Programming (OOP)

Example

```
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.

Example

```
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.

Example

```
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:

Example

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

Case-Sensitive

Variable names are case-sensitive.

Example

This will create two variables:

Object Oriented Programming (OOP)

```
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:

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

Example

Legal variable names:

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

Example

Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "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:

Object Oriented Programming (OOP)

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "John"
```

Many Values to Multiple Variables

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

Example

```
x, y, z = "Orange", "Banana", "Cherry"  
print(x)  
print(y)  
print(z)
```

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:

Example

```
x = y = z = "Orange"  
print(x)  
print(y)  
print(z)
```

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*.

Object Oriented Programming (OOP)

Example

Unpack a list:

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

Output Variables

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

Example

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

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

Example

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

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

Example

```
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:

Object Oriented Programming (OOP)

Example

```
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:

Example

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

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

Example

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

Python Data Types

Built-in Data Types

In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type: `str`

Numeric Types: `int, float, complex`

Sequence Types: `list, tuple, range`

Mapping Type: `dict`

Set Types: `set, frozenset`

Object Oriented Programming (OOP)

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:

Example

Print the data type of the variable x:

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

Python Numbers

There are three numeric types in Python:

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

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

Example

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

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

Example

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

Object Oriented Programming (OOP)

Int

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

Example

Integers:

```
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.

Example

Floats:

```
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.

Example

Floats:

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

print(type(x))
```

Object Oriented Programming (OOP)

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

Complex

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

Example

Complex:

```
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:

Example

Convert from one type to another:

```
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)
```

Object Oriented Programming (OOP)

```
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:

Example

Import the random module, and display a random number between 1 and 9:

```
import random  
  
print(random.randrange(1, 10))
```

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')
```

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)
```

Object Oriented Programming (OOP)

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)
```

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)
```

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])
```

Object Oriented Programming (OOP)

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)
```

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))
```

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)
```

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.")
```


Object Oriented Programming (OOP)

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)
```

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 String

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])
```

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])
```

Object Oriented Programming (OOP)

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:])
```

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])
```

Python - Modify Strings

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())
```

Lower Case

Example

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

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

Object Oriented Programming (OOP)

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!"
```

Replace String

Example

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

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!']
```

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)
```

Object Oriented Programming (OOP)

Example

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

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

String Format

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

Example

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

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))
```

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))
```

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

Object Oriented Programming (OOP)

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))
```

Boolean Values

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

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

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

Example

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

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

Example

Print a message based on whether the condition is **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,

Example

Evaluate a string and a number:

Object Oriented Programming (OOP)

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

Example

Evaluate two variables:

```
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.

Example

The following will return True:

```
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**.

Example

The following will return False:

```
bool(False)  
bool(None)  
bool(0)  
bool("")  
bool(()
```

Object Oriented Programming (OOP)

```
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**:

Example

```
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:

Example

Print the answer of a function:

```
def myFunction() :
    return True

print(myFunction())
```

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

Example

Print "YES!" if the function returns True, otherwise print "NO!":

```
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:

Object Oriented Programming (OOP)

Example

Check if an object is an integer or not:

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

Python Operators

Operators are used to perform operations on variables and values.

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

Example

```
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

Python Arithmetic Operators

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

Operator	Name	Example
+	Addition	x + y

Object Oriented Programming (OOP)

-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$

Object Oriented Programming (OOP)

<code>*=</code>	<code>x *= 3</code>	<code>x = x * 3</code>
<code>/=</code>	<code>x /= 3</code>	<code>x = x / 3</code>
<code>%=</code>	<code>x %= 3</code>	<code>x = x % 3</code>
<code>//=</code>	<code>x //= 3</code>	<code>x = x // 3</code>
<code>**=</code>	<code>x **= 3</code>	<code>x = x ** 3</code>
<code>&=</code>	<code>x &= 3</code>	<code>x = x & 3</code>
<code> =</code>	<code>x = 3</code>	<code>x = x 3</code>
<code>^=</code>	<code>x ^= 3</code>	<code>x = x ^ 3</code>
<code>>>=</code>	<code>x >>= 3</code>	<code>x = x >> 3</code>
<code><<=</code>	<code>x <<= 3</code>	<code>x = x << 3</code>

Python Comparison Operators

Comparison operators are used to compare two values:

Object Oriented Programming (OOP)

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

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4

Object Oriented Programming (OOP)

not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)
-----	---	-----------------------

Python Identity Operators

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

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

Python Membership Operators

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

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

Object Oriented Programming (OOP)

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

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

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: **a == b**
- Not Equals: **a != b**
- Less than: **a < b**
- Less than or equal to: **a <= b**
- Greater than: **a > b**
- Greater than or equal to: **a >= b**

Object Oriented Programming (OOP)

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

An "if statement" is written by using the **if** keyword.

Example

If statement:

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

In this example we use two variables, **a** and **b**, which are used as part of the if statement to test whether **b** is greater than **a**. As **a** is 33, and **b** is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
    print("b is greater than a") # you will get an error
```

Elif

The **elif** keyword is python's way of saying "if the previous conditions were not true, then try this condition".

Example

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

Object Oriented Programming (OOP)

```
print("a and b are equal")
```

In this example **a** is equal to **b**, so the first condition is not true, but the **elif** condition is true, so we print to screen that "a and b are equal".

Else

The **else** keyword catches anything which isn't caught by the preceding conditions.

Example

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

In this example **a** is greater than **b**, so the first condition is not true, also the **elif** condition is not true, so we go to the **else** condition and print to screen that "a is greater than b".

You can also have an **else** without the **elif**:

Example

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Object Oriented Programming (OOP)

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2  
b = 330  
print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
a = 330  
b = 330  
print("A") if a > b else print("=") if a ==  
b else print("B")
```

And

The **and** keyword is a logical operator, and is used to combine conditional statements:

Example

Test if **a** is greater than **b**, AND if **c** is greater than **a**:

```
a = 200  
b = 33  
c = 500  
if a > b and c > a:
```


Object Oriented Programming (OOP)

```
print("Both conditions are True")
```

Or

The **or** keyword is a logical operator, and is used to combine conditional statements:

Example

Test if **a** is greater than **b**, OR if **a** is greater than **c**:

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

Nested If

You can have **if** statements inside **if** statements, this is called *nested if* statements.

Example

```
x = 41

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

The pass Statement

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

Example

```
a = 33
b = 200
```

Object Oriented Programming (OOP)

```
if b > a:  
    pass
```

Python Loops

Python has two primitive loop commands:

- **while** loops
- **for** loops

The while Loop

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

Example

Print i as long as i is less than 6:

```
i = 1  
while i < 6:  
    print(i)  
    i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The **while** loop requires relevant variables to be ready, in this example we need to define an indexing variable, **i**, which we set to 1.

The break Statement

With the **break** statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1  
while i < 6:  
    print(i)  
    if i == 3:  
        break  
    i += 1
```

Object Oriented Programming (OOP)

The continue Statement

With the **continue** statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

The else Statement

With the **else** statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

Python For Loops

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

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

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

Object Oriented Programming (OOP)

Example

Print each fruit in a fruit list:

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

The **for** loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

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

The break Statement

With the **break** statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when **x** is "banana":

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

Example

Exit the loop when **x** is "banana", but this time the break comes before the print:

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

Object Oriented Programming (OOP)

The continue Statement

With the **continue** statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

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

The range() Function

To loop through a set of code a specified number of times, we can use the **range()** function,

The **range()** function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

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

Note that **range(6)** is not the values of 0 to 6, but the values 0 to 5.

The **range()** function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: **range(2, 6)**, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):
    print(x)
```

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

Object Oriented Programming (OOP)

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):  
    print(x)
```

Else in For Loop

The **else** keyword in a **for** loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

Note: The **else** block will NOT be executed if the loop is stopped by a **break** statement.

Example

Break the loop when **x** is 3, and see what happens with the **else** block:

```
for x in range(6):  
    if x == 3: break  
    print(x)  
else:  
    print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]
```

Object Oriented Programming (OOP)

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

The pass Statement

for loops cannot be empty, but if you for some reason have a **for** loop with no content, put in the **pass** statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:  
    pass
```

Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the **def** keyword:

Example

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():  
    print("Hello from a function")  
  
my_function()
```

Arguments

Information can be passed into functions as arguments.

Object Oriented Programming (OOP)

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):  
    print(fname + " Refsnes")  
  
my_function("Emil")  
my_function("Tobias")  
my_function("Linus")
```

Arguments are often shortened to *args* in Python documentations.

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
  
my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Object Oriented Programming (OOP)

Example

This function expects 2 arguments, but gets only 1:

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

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a `*` before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])  
  
my_function("Emil", "Tobias", "Linus")
```

Arbitrary Arguments are often shortened to **args* in Python documentations.

Keyword Arguments

You can also send arguments with the *key = value* syntax. This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
my_function(child1 = "Emil", child2 = "Tobias",  
            child3 = "Linus")
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

Object Oriented Programming (OOP)

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ****** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double ****** before the parameter name:

```
def my_function(**kid):  
    print("His last name is " + kid["lname"])  
  
my_function(fname = "Tobias", lname = "Refsnes")
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):  
    print("I am from " + country)  
  
my_function("Sweden")  
my_function("India")  
my_function()  
my_function("Brazil")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Object Oriented Programming (OOP)

Example

```
def my_function(food):  
    for x in food:  
        print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my_function(fruits)
```

Return Values

To let a function return a value, use the **return** statement:

Example

```
def my_function(x):  
    return 5 * x  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a **function** definition with no content, put in the **pass** statement to avoid getting an error.

Example

```
def myfunction():  
    pass
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses

Object Oriented Programming (OOP)

excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

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

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

Python Lambda

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

Syntax

lambda arguments : expression

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

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

Lambda functions can take any number of arguments:

Object Oriented Programming (OOP)

Example

Multiply argument **a** with argument **b** and return the result:

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

Example

Summarize argument **a**, **b**, and **c** and return the result:

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

Why Use Lambda Functions?

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

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

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

Use that function definition to make a function that always doubles the number you send in:

Example

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

```
mydoubler = myfunc(2)
```

```
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

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

```
mytripler = myfunc(3)
```

Object Oriented Programming (OOP)

```
print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

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

```
mydoubler = myfunc(2)  
mytripler = myfunc(3)
```

```
print(mydoubler(11))  
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Arrays

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

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"  
car2 = "Volvo"  
car3 = "BMW"
```

Object Oriented Programming (OOP)

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

Note: The length of an array is always one more than the highest array index.

Looping Array Elements

You can use the `for in` loop to loop through all the elements of an array.

Example

Print each item in the `cars` array:

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

Object Oriented Programming (OOP)

Adding Array Elements

You can use the **append()** method to add an element to an array.

Example

Add one more element to the **cars** array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the **pop()** method to remove an element from the array.

Example

Delete the second element of the **cars** array:

```
cars.pop(1)
```

You can also use the **remove()** method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

Note: The list's **remove()** method only removes the first occurrence of the specified value.

Array Methods

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

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list

Object Oriented Programming (OOP)

<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

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