# OOSD Content

# Week 1

## Introduction

### Software development

#### What is software?

The word software denotes instructions for a machine, organised into helping humans use the machine. The physical machine (the hardware) can be anything, but to be controlled by software it must include a microprocessor.

#### Why not just have the machine and use it? Why software?

Every machine's functionality is embodied in its hardware, but some machines are more limited than others. An old-fashioned hairdryer blows air at two or three heat settings and that is all. Controls can be implemented using fairly simple electrical circuits.   
  
A programmable washing machine is also limited in what it can do but there is scope for variation. To allow users to configure a washing machine, an electronic circuit (pure hardware) could be used, but it this would have to be more complicated than the simple circuits of the hairdryer. The alternative is to have a processor and software. The processor consists of circuits that perform a limited number of primitive operations and software is a recipe for how these primitive operations should be combined to achieve something. For all but very simple functionality, this is **cheaper to produce than a specialised electronic circuit**.   
  
However, the true value of this setup (processor + software) lies in another property it has. While a washing machine can only wash clothes (admittedly in several different ways), less specialised hardware and a more complex user interface can provide user-defined functionality through the **theoretically infinite combinatorial power of sofware as a recipe**. This is what computers and smartphones do. The versatility of their user interfaces allows for ever new functionality to be invented and their internal programmabiliy supports its implementation.

#### What is software physically?

A unit of software usable for some purpose is called a software program, application, app or product. One of these is like the contents of a book, e.g. Oliver Twist by Charles Dickens. When we call out that name we generally do not mean any particular paper or electronic copy of the story, we mean the words that Dickens wrote and that we read. It is like that with software, which, although contained in a file on disk or in the memory of a device, cannot be equated with where it is stored but consists of abstract instructional content, which can be copied and stored in multiple places at once. **It is not physical.**

#### How does software development fit into all this?

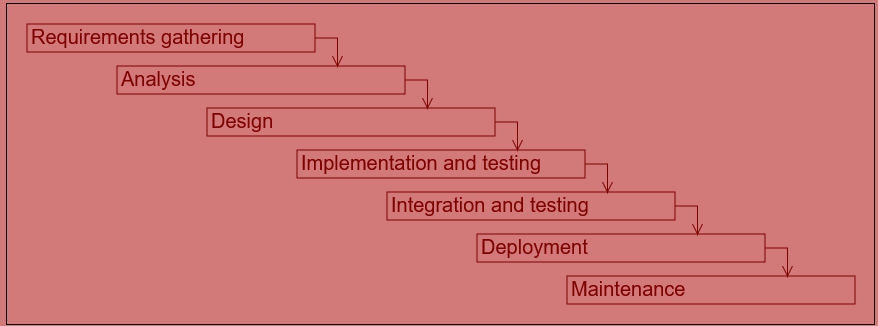
Putting together a program does not require listing of primitive instructions understood by a processor. These are packed into higher level constructs and ready-made software chunks of a programming language, which is less granular and more user-friendly for humans. Different programming languages present the underlying processor capabilities in different ways, depending on their purpose but also simply on the preferences of their designers.

What is the difference between software development, programming and coding?

Different people and even different dictionaries will give you different definitions, with the distinction between the terms varying from huge to none. However, when a distinction is made, software development is thought of as broader than programming, which in turn is treated as broader than coding. So, coding is simply writing the instructions (or code) in the language; programming involves a bit more e.g. design and testing but is still performed by one person; software development is the entire process that starts with an idea and ends in a product and could involve many people and organisations. Software development can also be used as a term for the entire industry involved in producing software.

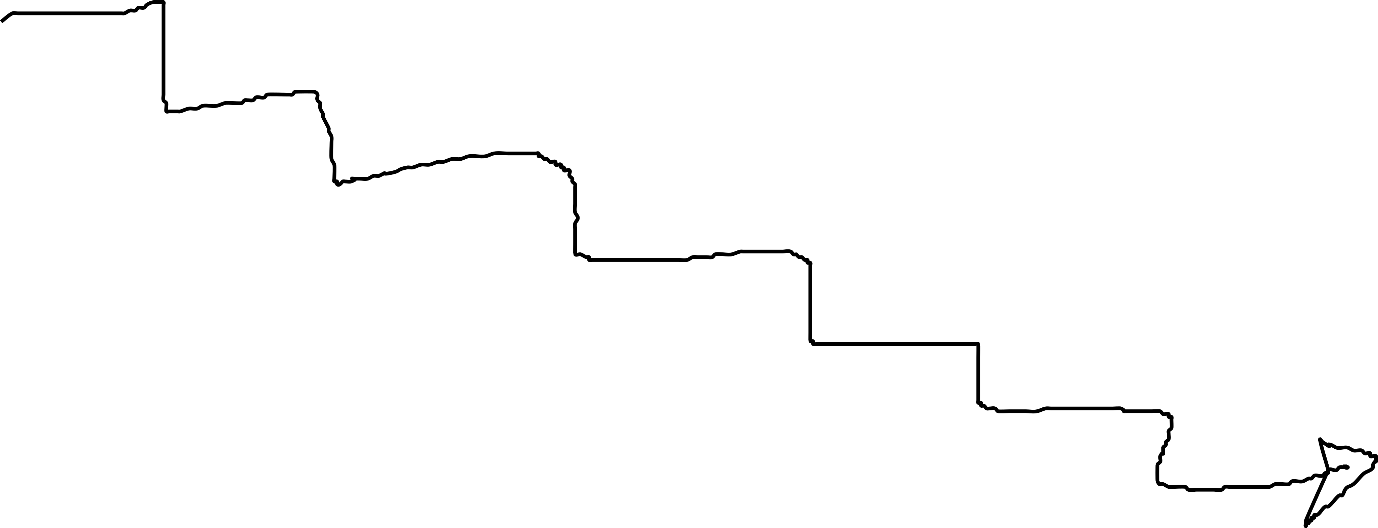
### Methodologies

Developing software is a complex undertaking that includes several different types of activity within its lifecycle (the software development lifecycle - SDLC):



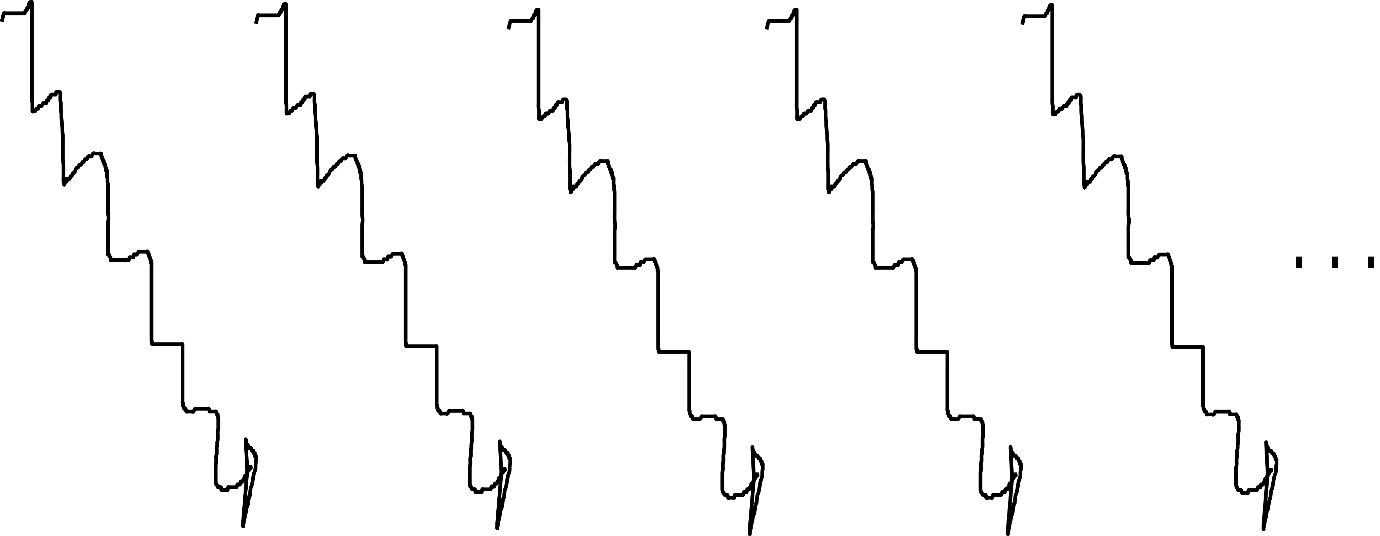
Depending on the context, the way these activities are interconnected and carried out as well as the degree of formality employed will differ greatly. However a **methodology** will generally be adhered to in one form or another.

The two main methodology styles are:



#### waterfall

* + projects are large
  + each activity is a single phase in a large project
  + phases last weeks or months
  + a project generally corresponds to a single release



#### agile (e.g. Scrum, XP)

* + projects consist of many small iterations (1-2 weeks)
  + constant updating of requirements based on feedback
  + constant communication between teams and stakeholders (internal and external)
  + continuous delivery and even deployment
  + goes hand-in-hand with DevOps (also promotes connectedness of activities and roles but is centered around *delivery* as opposed to *development* in agile, see [this](https://www.guru99.com/agile-vs-devops.html))

But there are many more that are either variations on these two themes or sit somewhere between the two, for example:

#### Rapid Application Development (RAD)

* + less focus on design and documentation
  + prototyping
  + multiple iterations

#### 'Sashimi' model - with overlapping phases

### Methods and tools

There are methods and tools for every activity in the software development cycle.

#### requirements gathering and analysis,

* using tools such as Eclipse-based Rational tools or VisualParadigm
  + use cases e.g. as prescribed by Rational Unified Process (RUP) framework
  + user stories in agile
  + source-stimulus-environment-artefact-response-measure for non-functional requirements
  + analysis-level class diagrams

#### design, using tools such as Eclipse-based Rational tools or VisualParadigm

* + component diagrams (describing high-level structure)
  + design-level class diagrams (describing structure)
  + sequence diagrams (describing behaviour)

#### implementation

* + based around programming in languages such as Python, Java, C++
  + supported by many different types of tools:
    - syntax-aware text editors e.g. Visual Studio Code, Notepad++
    - interpreters and compilers for
    - debuggers
    - program profilers and validators e.g. Valgrind
    - version control systems e.g. git implemented by GitHub, Bitbucket etc.
    - integrated development environments (IDEs) e.g. Eclipse, NetBeans, PyCharm; providing integrated access to all of the above listed tools, typically
      * text editor
      * compiler/interpreter
      * debugger
      * profiler
      * version control

#### integration, deployment, maintenance

* + tools for the automation of configuration and deployment such as Puppet, Chef, Ansible

#### planning and management (orthogonal to all of the above) also require tools for

* + defect tracking
  + project management (traditional)
  + agile tools, which include both of the above

### The substance of software and programming languages

Software has two aspects, both equally important:

* **information**, represented in a program by **data structures**
* **behaviour**, represented in a program by **algorithms**

A **programming language** consists of vocabulary, syntax and rules for representing *information* and *behaviour*.

Here are some broad classifications of programming languages:

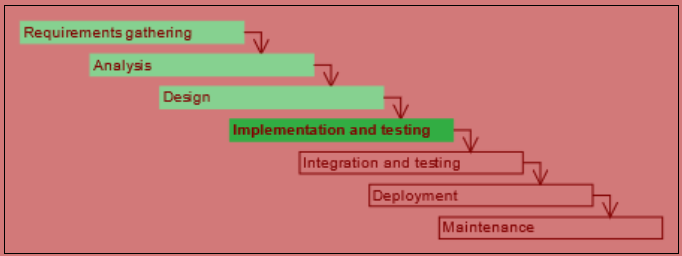
* imperative (Java) vs. declarative (SQL), by focus, on controlling behaviour vs. information
* object-oriented (C++) vs. procedural (C), by how behaviour and information are organised
* high-level (e.g. C++) vs. low-level (e.g. assembly), by closeness to the processor
* compiled (e.g. C++) vs. interpreted (e.g. Python), by mode of translation, in advance as a whole vs. during execution
* special-purpose (e.g. R for data analysis) vs. general-purpose (e.g. Python)

### What does the Object Oriented Software Development module cover?

* This module focuses primarily on
  + **implementation and testing**

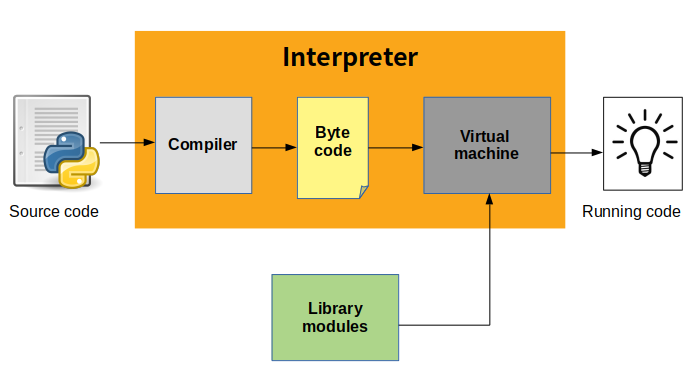
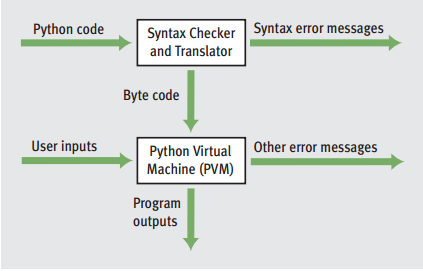
but also provides an introduction to

* + requirements gathering
  + analysis
  + design



* In the module we employ **problem-based learning (PBL)**, which engages the student in learning through solving open-ended problems. This is particularly suited to software development as it mirrors the real-life (workplace and advanced study) circumstances of software development application.
* The module is taught through **Python**, a high-level interpreted general-purpose programming language.

### Python and tools

* The Python programming language can be used
  + interactively on the command line for simple work
  + in one or more interacting scripts that can be kicked off to execute as a unit
* Python is an interpreted language, requiring no explicit compilation step on the part of the developer (traditionally this type of language would have been used for scripting). Python interpreter  
  (source: indianpythonista.wordpress.com) 
* A program in Python (or any other language) is useful only if it interacts with its environment by consuming and providing information Python program inputs and outputs  
  (source: python-nitol.blogspot.com) 

## To Begin

### Functions

Functions in Python are **chunks of functionality** that can be called to do something. The list of Python functions available with the interpreter can be found [**here**](https://docs.python.org/3/library/functions.html). We will be looking at some of these later.

#### Functions can be

* part of the standard library that is installed with the Python interpreter
* part of **modules** or **packages** that can be added to the installation
* defined and used in a newly developed Python program

For example, to exit the Python interactive interpreter, function **exit()** or **quit()** can be used.

Functions for exiting the interactive interpreter (only available in interactive mode) [CS-1]

exit()

e# or

quit()

#### More about functions:

* Function calls are denoted with **parentheses following the function name**. The parentheses may be empty or contain arguments
* Functions **can take one or more arguments**, i.e. pieces of information that they use when carrying out whatever they are designed to do. The call to the function print() in code snippet [[CS-2]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_1_begin.html#py-func-print) is made with argument 3. The function print() prints the argument passed to it to the console.

A function with an argument (run interactively) [CS-2]

print(3)

* Functions **can return values** and this is often their purpose (to create a value by calculating, converting or performing some other action that results in some data).

A function that returns a value (run interactively) [CS-3]

round(4.3)

round(4.123456454323455675544, 4)

Functions will be introduced and used throughout the module. The syllabus, however, does not cover the definition of own functions.

### The Help Function

While Python documentation is available on the [**Python documents page**](https://www.python.org/doc/), it is also possible to get information on individual items programmatically, using the function help().

Using the help() function (run interactively) [CS-4]

help(str)

help(True)

help(help)

### Objects and methods

**Roughly anything of substance,** i.e. **anything that has a value and a type**, is an **object** in Python. Thus a number or a piece of text is an object, as are more complicated structures with many pieces of individual information.

A function can be associated with a particular type of object and in that case it is called a **method**. Methods are called with the syntax

<object name or value>.<method\_name>()

The example in code snippet [[CS-5]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_1_begin.html#py-method-example) is that of the method upper called on an object containing the text "hello". Try it out.

Example of a method (run interactively) [CS-5]

"hello".upper()

### Comments

# This line will be ignored by the interpeter

exit() # exit() runs but his part of the line is ignored

### Imports

The core functionality of Python is available in the interpreter by virtue of Python having been installed.

Some functionality will have been installed but needs to be 'brought in' with a module import.

Module import (run interactively) [CS-7]

# generate random number between 1 and 10

random.randint(1, 10) # error!

# the random module must be imported

import random

random.randint(1, 10)

Some more specialised packages of functionality will not be installed with Python (this varies between operating systems) and in that case packages will need to be installed before contained modules can be successfully imported.

## Data Types and Operators

### Data Types and Operators

Python defines many data types. In this module we will be using only some of them, namely, boolean, integer, float, string, list and range.

The function [**type()**](https://docs.python.org/3/library/functions.html#type) can be used to check the type of a piece of data

#### Boolean datatype

* A piece of data of type *boolean* carries a *truth value*
* Possible values:

True

False

Note that True and False are written with capitalised first letter and represent *keywords* in Python.

#### Logical Operators

* Operate over *boolean* values
* List (exhaustive):

**and**

binary operation, both operands must be true for the result to be true

**or**

binary operation, at least one of the two operands must be true for the result to be true

**not**

unary operation, result is the opposite of the operand

Logical operators (run interactively) [CS-1]

*type(True)*

*type(False)*

*True and True*

*True and False*

*False or False*

*False or True*

*not True*

*not False*

#### Integers

* A piece of data of type *integer* represents a **whole number**
* Any whole number (e.g. 0, 22, -12345) can be represented as an integer in Python

Python 2 had limits as to the integer values that it could represent. Python 3 theoretically can represent any whole number, with limits imposed only by the memory of the computer it runs on.

Very large integers (run interactively) [CS-2]

*type(123)*

*10\*\*100 # on my computer this instantly returns the number (called a googol)*

*10\*\*1000000 # on my computer this returns the number but after about 10 seconds*

#### Floats

* A piece of data of type *float* represents **real number**
* Because of the way that floats are represented in most hardware (using double-precision binary fractions)
  + up to *15 significant digits* can be represented Float rounding (run interactively) [CS-3]
  + type(4)
  + type(4.0)
  + # the value is implicitly rounded before the expression is written out
  + 0.12345678901234567890
  + # rounding to 2 digits
  + round(0.12345678901234567890, 2)
  + # rounding to 10 digits
  + round(0.12345678901234567890, 10)
  + # rounding to 18 digits - some digits are not representable, implicit rounding

round(0.12345678901234567890, 18)

In the code snippet above we have used a new function, [**round()**](https://docs.python.org/3/library/functions.html#round).

* + comparisons of floats may not yield the correct answer (see examples in comparison operator section below)

#### Arithmetic operators

* Operate over *integer* and *float* values
* List (non-exhaustive, visit [**the Python Library Reference**](https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex) for a full list):

+

addition

-

subtraction

\*

multiplication

/

division (yields a float result)

//

integer division (yields an integer)

%

modulo (yields the remainder)

\*\*

power

Arithmetic operators (run interactively) [CS-4]

3 + 4

7 / 2

7 // 2

2 \*\* 3

#### Comparison operators

* Comparison operators include equality and relational operators.
* Operate over almost any types where an interpretation is possible.
* List (non-exhaustive, visit [**the Python Library Reference**](https://docs.python.org/3/library/stdtypes.html#comparisons) for a full list):

<

less than

>

greater than

<=

less than or equal to

>=

greater than or equal to

==

equal to

!=

not equal to

With the number types, these operators generally work as expected.

Comparison operators (run interactively) [CS-5]

3 < 4

3 == 3

4.0 != 5.5

However, there are some effects stemming from the way floats are internally represented (as base-2 fractions, with a limited number of bits) that we need to watch out for. For most real number values, their internal representation will be slightly 'off'. This means that when adding two real numbers there is no guarantee that the sum of representations will be equal to the representation of the sum. This effect will only appear outside of the range of 'guaranteed' consistent significant digits.

Operations with floats to watch out for (run interactively) [CS-6]

# the following gives an unexpected result

0.2 + 0.1 == 0.3

# because the value is not what we expected either

0.2 + 0.1

# always round before using the equal operator on floats

round(0.2 + 0.1, 15) == round(0.3, 15)

#### Strings

* In Python, a string is an ordered sequence of characters encoded using UTF-8 and representing some **textual data**.
* The string type is called str.
* str is a **sequence** type, along with list, tuple and range.
* A character on its own is not a 'thing' in Python. A character is represented as a string of length 1.
* A string can be of any length, including 0. Getting the length of a string (run interactively) [CS-7]
* len("Bob")

len("construction")

* String values are denoted using quotes (single or double), with the characters of the string placed between them.
* Characters such as new lines and tabs can be included in a string. These must represented with the appropriate **escape sequence**: '\n' for a new line and '\t' for a tab. Other escape sequences that we will use a lot are those for representing quotes within quotes. For reference, you might like to have a look at a [**complete list of escape sequences**](https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals) (you need to scroll down a bit) Using escape characters (run interactively) [CS-8]
* print("He looked around and said \"Interesting\"")
* print("one line\nanother line\thop\thop again")

Notice that the escape sequences for new line and tab remain intact when the string is displayed. A string needs to be written to a file or screen for these characters to be rendered. We will see them rendered in section 'Printing'.

#### String concatenation

There are many useful string operations in Python, but for the moment we will look at concatenation using the +. We can concatenate two or more different strings or 'multiply' the same string.

String concatenation with + and \* (run interactively) [CS-9]

"hello" + " " + "there"

"bye bye" + ' ' + 'here'

"blah " \* 3

### Casting between types

Python provides functions for converting pieces of data from one type to another.

int()

converts any type to an integer (if possible)

float()

converts any type to a float (if possible)

str()

converts any type to a string

bool()

converts any type to a bool

Here is some general information on what to expect when casting, but when in doubt check the documentation or try it!

Almost any piece of data can be converted to a boolean:

for numeric data types, a value of zero is converted to False and everything else to True

for strings, an empty string is converted to False and any other string to True

With the flexibility of text, strings are also an easy type to convert to from anything else.

Booleans are easily converted to numeric types, but strings can be converted only if they express a number through text.

Casting (run interactively) [CS-10]

bool(34)

bool(0)

bool("")

bool("abc")

int(4.3)

int("4.3")

int("4.3abc")

float("3")

float(True)

str(2345)

str(True)

### The value and type of expressions

**An expression** in Python is an executable construct that when executed results in **a value with a type** i.e. can be evaluated. Expressions can be used as pieces of data, in all the ways that literal values, such as 5 or "Some text", can be used.

**Example 1**

The interactive interpreter console evaluates any expression passed to it and displays the value obtained (in batch mode, this automatic displaying of values does not occur and they must be explicitly shown if needed, e.g. with print()).

Expression value and type (run interactively) [CS-11]

# an expression's value will be displayed

3 + 4

# display the expression's type

type(3 + 4)

**Example 2**

A function call is also an expression and it evaluates to the returned value. The casting functions are a clear example of this. For example, int("34") returns the integer 34 and thus evaluates to 34. If we make this call inside the interactive interpreter, 34 is printed as this is the value of the expression.

Checking the types of casting function calls (run interactively) [CS-12]

type(bool("abc"))

type(int(4.3))

type(float("3"))

type(str(True))

**Example 3**

The function call print("a") is not really an expression, as it does not return a value and thus does not evaluate to anything. However, if we try to treat it as an expression, checking its value and type, we do not get an error but the value None and type NoneType are returned. Functions that do not return a value could thus be thought of as quasi-expressions. This differentiates them from statements, the other type of executable language construct, which represent *actions* rather than *things* and cannot be evaluated. Statements will be treated later in this module.

Value and type of print() [CS-13]

# the function call

print("a")

# force display of function call expression value

print(print("a"))

# display the function call expression type

type(print("a"))

### Printing

The print() function writes to the console. We have encountered it already but here we have a closer look at it through examples.

* print() can take any number of arguments of different types, separated by commas The print() function - basic use [CS-14]
* print(345)
* print("Hello there")
* print(True)
* print("Hello,", "Bob")

print('I am', 100, 'years old')

In the output of multi-argument calls, a whitespace character separates the printed arguments.

* When printing, escape characters for new lines and tabs ('\n' and '\t') are rendered ([**list of escape sequences**](https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals), you need to scroll down to see the table). Using escape characters (run interactively) [CS-15]

print("one line\nanother line\thop\thop again")

* By default, print() ends the output with a new line. This can be changed with the use of a **named argument**, end (see section below, 'Function arguments revisited"). Any string can be specified as the terminating string of the print output. Using the end argument (run interactively) [CS-16]
* print("This is followed by a new line")
* print("This is followed by a full stop", end=".")

print("This is followed by nonsense", end="\tas;dlkfj;alkjksjdflkdjflkj")

## Variables

So far we have learnt about data types and how pieces of data can be manipulated, individually or in groups. However, the functionality that we have used for this has all been Python built-in functionality and the instances of data manipulation carried out have been tied to particular fixed data values or **literals**. We have more or less used Python as a sophisticated kind of calculator.

To create some reusable functionality, in most cases we need to be able to manipulate **placeholders**, rather than only known data values.

In class exercise!! >>> [CE-A] Can you think of two cases where a useful program can be defined without placeholders i.e. work without inputs from the outside world?

There are broadly two types of placeholders in programs:

* **function parameters** (the pieces of data that a function says it accepts), which are used as placeholders for defining what the function does and are replaced by arguments (actual pieces of data) when the function is called
* **variables**, which are, within a context, standalone placeholders for pieces of data

Writing programs using only functions, with function parameters the only kind of placeholder, is called *functional programming* and can be pursued in Python (although it is not optimised for it), but this is not what we are going to do (especially since funcitions are not even on the syllabus!). So let's look at variables.

Python has a very simple approach to variables. They are **named** and **given a value** in one expression, which can be placed anywhere in the code.

**Variable naming and value assignment**

* a variable name
  + can include only alphanumeric characters and underscores ('\_')
  + must not start with a number
  + must not be a [**Python keyword**](https://docs.python.org/3/reference/lexical_analysis.html#keywords)
  + using any built-in function names (e.g. str or len) as variable names would make the functions unavailable and needs to be avoided; you can check if something is a built in function by typing in the name into the Python command line Checking if names are used already [CS-1]
  + len
  + # expected output:
  + <built-in function len>
  + str
  + # expected output:
  + <class 'str'>
  + xyz # a string that is not already used
  + # expected output:
  + Traceback (most recent call last):
  + File "<stdin>", line 1, in
  + NameError: name 'xyz' is not defined

# we can proceed to use xyz as a variable name

* the assignment is made with the **assignment operator =** (this is different from the equality operator, ==, which tests values for equality)
* the assigned value can be of any type
* if a new value is assigned to the same name further in the program flow, the variable takes on the new value and its type
* the value assigned to a variable can be that of a *literal*, of *another variable* or of a *more complex expression*

Naming a variable and giving it a value (run interactively) [CS-2]

# setting variables with literal values

number = 3

a\_decimal\_number = 3.3

myname = "Jane"

isItTrue = True

r2d2Status = 'active'

# setting variables from other variables

ano0therNum = number

C3POStatus = r2d2Status

# some invalid names:

True = "the value of true"

2bOrNot2b = 22

# setting a new value to an existing variable

a\_decimal\_number = 5.5

**Variable use**

* a variable must be named and set (given a value) before it is used in the program flow
* a variable is used in the same way that a literal value would be used, for example to provide the value of another variable

Using variables (run interactively, after [[CS-2]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_3_variables.html#py-var-create)) [CS-3]

# display values in the Python console

a\_decimal\_number

C3POStatus

# add two numbers

number + a\_decimal\_number

# create a new variable and print it

status = "Like r2d2, " + myname + " is " + r2d2Status

print(status)

**Assignment operators**

Apart from the standard assignment operator, =, there are the **augmented assignment** operators, which combine another operation with assignment: +=, -=, \*=, /= etc.

For example, x += 3 means the same as x = x + 3, but is generally more efficiently implemented under the hood.

Augmented assignment operators (run interactively) [CS-4]

x = 3

x += 4

print(x)

name = "John"

name += " Doe"

print(name)

### Input

One way of setting variables that we will use a lot in this module is input from the console. At this beginner level of programming, reading data from the console, in conjunction with printing, is an easy way to achieve two-way communication with the outside world and make a program useful.

When the function input() is called, it waits for a line to be typed in and the Enter key pressed on the console. Then it returns the text contained in the submitted line, as a string value.

Input (run as script) [CS-5]

print("Enter some text:")

input()

# enter text on console, press Enter

print("Enter your name:")

name = input()

# enter name on console, press Enter

print("Your name is", name)

The code using input() in [[CS-5]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_3_variables.html#py-input) can be shortened by using an argument to input(). This argument is printed to prompt the user, rather than relying on a separate call to print().

Input with argument (run as script) [CS-6]

input("Enter some text: ")

# enter text on console, press Enter

name = input("Enter your name: ")

# enter name on console, press Enter

print("Your name is", name)

It is important to remember that the value returned by input() is always a string. If a different type is required, a conversion needs to take place.

Input with type conversion (run as script) [CS-7]

x = input("Enter a whole number: ")

y = input("Enter another whole number: ")

# x and y are both of type string

print("The sum of your numbers without conversion is", x + y)

# int(x) and int(y) will be integers and so will their sum

print("The sum of your numbers with conversion is", int(x) + int(y))

# Week 2

## Strings as objects

The string type, str, is a **sequence** type in Python (along with *list*, *tuple* and *range*).

As with all other types in Python, instances of str are *objects*. Objects and methods were [briefly introduced](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_1_begin.html#ooswd-l1-03) at the beginning.

### Some string methods

Here we will look at some methods that can be called on a string object. The full list can be found [**here**](https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str).

* A very useful way of constructing strings is to use the [**format()**](https://docs.python.org/3/tutorial/inputoutput.html#the-string-format-method) method (methods are like functions, but are associated with a piece of data and called 'on it'). This particular method allows inserting values of any type into other strings. String format (run interactively) [CS-1]

"Hello {0}, it must be {1} years since we last met!".format("Bob", 7)

* There are various useful text modification string methods. Various text modification methods (run interactively) [CS-2]
* # call on literal
* "hello".upper()
* # call on variable
* name = "tyrannosaurus rex"
* name.upper()
* name.capitalize()
* name.isnumeric()

"234".isnumeric()

**Operators**

We have already seen the concatenation operators + and \*, but there are others that can be used with the string type. We will look at some of them but a full list is available [**here**](https://docs.python.org/3/library/stdtypes.html#common-sequence-operations).

Behind the scene, operators are implemented as methods. If you run help(str) in the interactive interpreter to get information on the string type, you will notice a number of methods listed at the beginning that start and end with double underscores (\_\_). These methods are not intended to be called directly but are implementations of operators. For example, the method \_\_contains\_\_() will be called when the operator in is used. We can use both operator and method to the same effect, **but it is the operator that should be used**.

#### Operator in

The return value of this operator is of type **boolean**.

Checking for a substring with the in operator (run interactively) [CS-3]

dinosaur = "tyrannosaurus rex"

"ran" in dinosaur

"swam" in dinosaur

# method works but should not be used!

dinosaur.\_\_contains\_\_("ran")

dinosaur.\_\_contains\_\_("swam")

#### Comparison

The return value of this operator is of type **boolean**.

Comparing strings (run interactively) [CS-4]

name1 = "Jane"

name2 = "Jack"

name1 == name2

name1 < name2

"." < ":"

In code snippet [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_4_strings.html#py-str-comparison) the result of the last expression may be surprising, but will be explained in the next section, 'UTF-8'.

#### Operator []

Square brackets can be used to read an individual character or a range of characters from a string (slicing).

The return value of this operator is a **string**.

some\_string[i]

get a single character at index i

some\_string[i:j]

get a substring starting at index i and ending before index j

some\_string[:j]

get a substring starting at the beginning of the original string and ending before index j

some\_string[i:]

get a substring starting with the character at index i and containing the remainder of the original string

Accessing a substring using square brackets (run interactively) [CS-5]

name = "Charlie"

name[0]

name[3]

name[2:4]

name[4:]

name[:3]

name[:]

name[:3] + name[5]

Negative indices can be used to access characters and slices in a string. The last character is in negative position -1, the second last in position -2 and so on.

Accessing a substrings with negative indices (run interactively) [CS-6]

name = "Charlie"

# these are equivalent to the first 5 expressions in [[CS-5]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_4_strings.html#py-str-sq-brack)

name[-7]

name[-4]

name[-5:-3]

name[-3:]

name[:-4]

#### Immutability

Strings are **immutable** objects, which means that a string value cannot be changed 'in place'. A string can be read and its value or parts thereof used as a basis for the creation of new strings, but these will be stored in a different memory location from the original string, the original string remaining intact.

Replacing substrings to demonstrate immutability (run interactively) [CS-7]

name = "Johnathan"

shorter = name.replace("natha", "")

name

shorter

In code snippet [[CS-7]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_4_strings.html#py-str-immut) a call is made on the variable name for the string "natha" to be replaced with an empty string "". The function does not modify the string stored in variable name (this would not be allowed because strings are immutable) but creates a new string with the new value and returns that new value. We store it in variable shorter.

### UTF-8 Encoding

In Python 3 strings are encoded using the UTF-8 character encoding system. An encoding system maps the visual appearance of characters to numeric values that can be stored and manipulated in a computer. A full list of characters mapped by UTF-8 can be found [**here**](https://mcdlr.com/utf-8).

* When strings are compared, what is really compared is the UTF-8 numbers that underlie them.
* The functions ord() and chr() convert between the character and an integer representing its UTF-8 code. Converting between characters and their UTF-8 codes (run interactively) [CS-8]
* # ord() must be called on a string of length 1
* ord("a")
* ord("abc") # cases an error
* chr(45)
* chr(ord("a"))

ord('ć')

* The first 128 characters in the UTF-8 mapping are exactly the same as the ASCII encoding map. ASCII (from American Standard Code for Information Interchange) was designed in the 1960s and widely used in computing ever since. Backward compatibility is thus important and UTF-8 provides it.

In code snippet [[CS-8]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_4_strings.html#py-str-utf8) the function call ord('a') yields 97, which is in the ASCII range. The call ord('ć') yields 263, which is outside of the ASCII range.

Write a short program that takes two inputs: a character and an integer (call it an encryption key). The program should then encrypt the character into another character and print this new character out. The encryption algorithm is as follows: add the encryption key to the character's UTF-8 code to get the UTF-8 code of the character's encrypted value.

[EXAMPLE SOLUTION](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/strings/str_encoding.py)

String comparison cannot be used directly for **alphabetical ordering**. Consider the example in code snippet [[CS-9]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_4_strings.html#py-str-alpha-ord). Lower case 'a' is considered to be greater than upper case 'a', because the UTF-8 code for the upper case letter is lower, but for alphabetical ordering we would like all letters 'a' to have the same value, regarless of case. This can be solved by converting all the text to either upper case or lower case before comparison.

Alphabetical ordering of strings (run interactively) [CS-9]

'a' > 'A'

# the above is true because

ord('a') > ord('A')

name = 'Jack'

name\_again = 'jack'

name == name\_again

# the above yields False

# alphabetical ordering

name.lower() == name\_again.lower()