# 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?

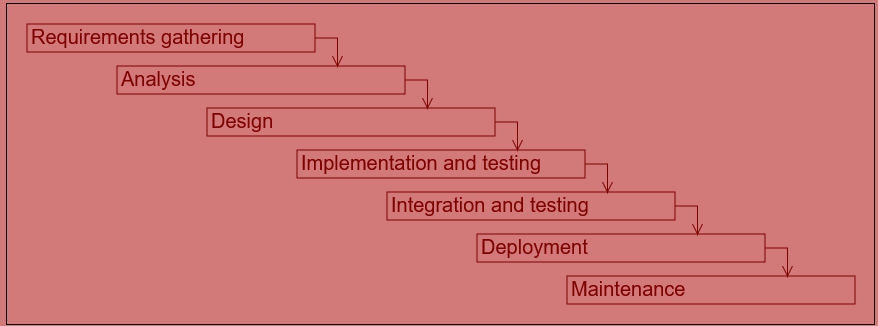
Putting together a program does not require listing of primitive instructions understood by a processor. These are packed into higher level constructs and ready-made software chunks of a programming language, which is less granular and more user-friendly for humans. Different programming languages present 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.

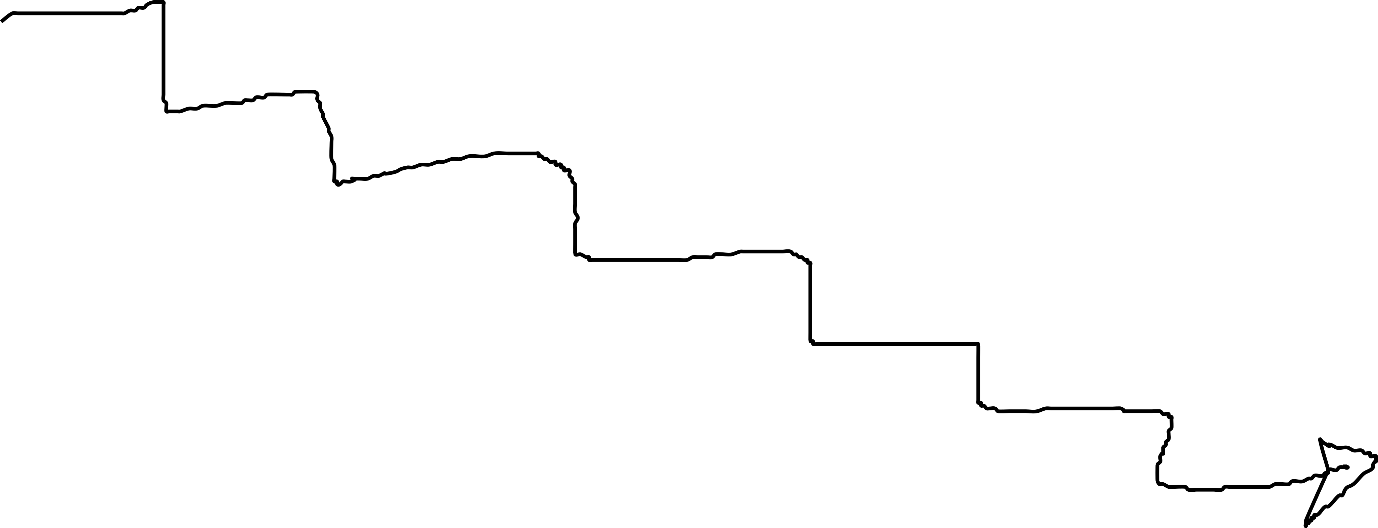
### Methodologies

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



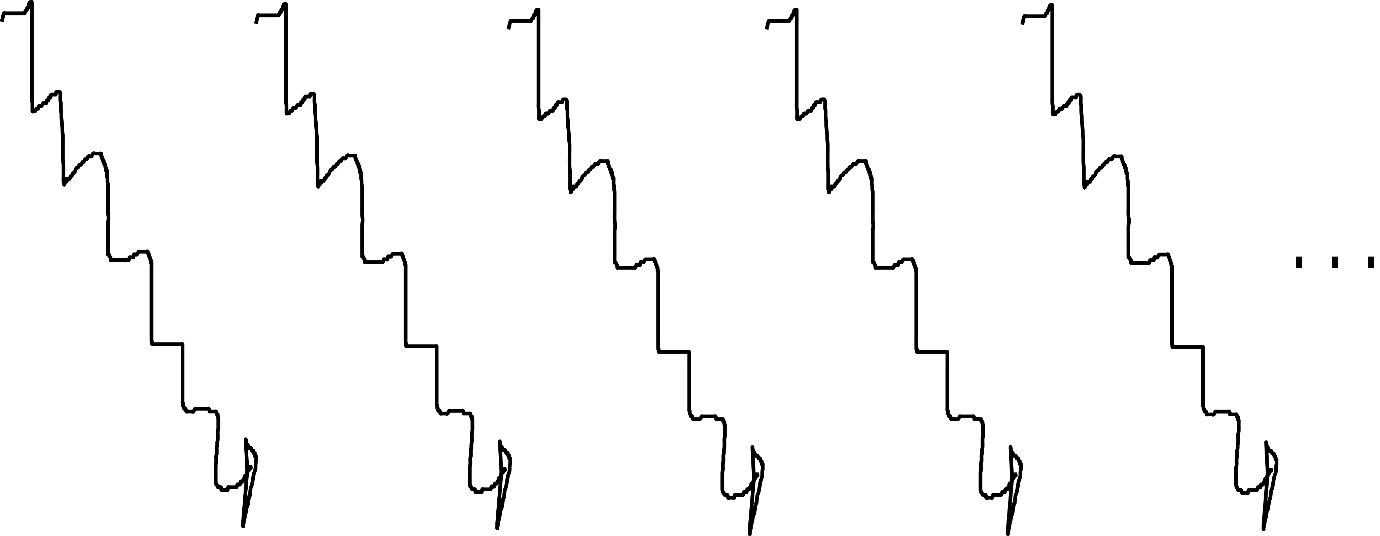
Depending on the context, the way these activities are interconnected and carried out as well as the degree of formality employed will differ greatly. However a **methodology** will generally be adhered to in one form or another.

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:

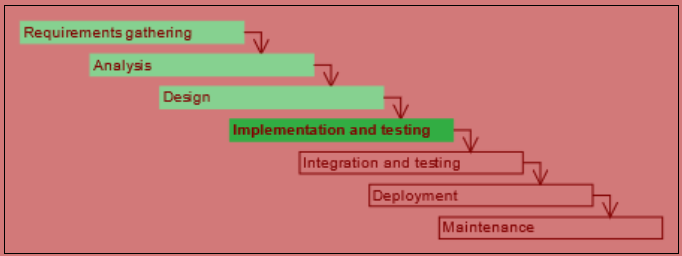
* imperative (Java) vs. declarative (SQL), by focus, on controlling behaviour vs. information
* object-oriented (C++) vs. procedural (C), by how behaviour and information are organised
* high-level (e.g. C++) vs. low-level (e.g. assembly), by closeness to the processor
* compiled (e.g. C++) vs. interpreted (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**

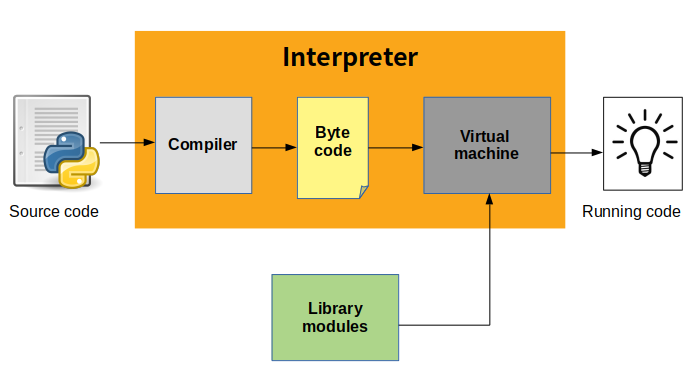
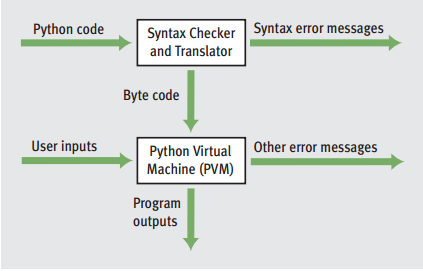
but also provides an introduction to

* + 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.

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

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.

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

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).

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

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().

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

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

**"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.

##### Example [CS-7] - Module import (run interactively)

**# 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

##### Example [CS-1] - Logical operators (run interactively)

*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.

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

***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

##### Example [CS-3] - Float rounding (run interactively)

**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

##### Example [CS-4] - Arithmetic operators (run interactively)

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.

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

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.

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

# 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().

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

**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.

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

**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

### 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.

##### Example [CS-1[ - String format (run interactively)

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

* There are various useful text modification string methods.

##### Example [CS-2] - Various text modification methods (run interactively)

**# 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**.

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

**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**.

###### Example [CS-4] Comparing strings (run interactively)

**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

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

**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.

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

**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.

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

**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.

##### Example [ CS-8] - Converting between characters and their UTF-8 codes (run interactively)

**# 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.

#### In-class exercise[CE-A]

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.

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

**'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()**

## Lists

### What is a list in Python?

* A list is an ordered collection of objects of any type. A list is itself an object.
* The list type in Python is called list.
* list is a sequence type, along with **str**, **tuple** and **range.**
* A list literal is denoted using square brackets, with the elements of the list specified inside the brackets and separated by commas. The elements can be literals or variables.
* A list can be empty.
* A list can contain another list.

Unlike strings**, lists are mutable objects in Python**, which means that their value can be modified 'in place' i.e. at the memory location where they are stored.

##### Example [CS-1] - Defining lists (run interactively)

**a\_string = "hohoho"**

**an\_int = 12345**

**a\_list = ["hello", 4, a\_string, 5.6, "a", an\_int, True]**

**type(a\_list)**

**empty\_list = []**

**listWithListElement = ["hi", ["Jane", "Doe", 21], True, 55.3]**

### List operations and methods

As lists are mutable, they support both [**operations that do not change the sequence object**](https://docs.python.org/3/library/stdtypes.html#common-sequence-operations) and [**those that do**](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types).

#### Operations that do not change the list object

These operations are the ones also supported by the string sequence type.

##### Concatenation (using operators + and \*)

Concatenation does not modify the lists that are being concatenated but returns a new list.

###### Example [CS-2] - List concatenation (run interactively)

**["a", "b"] + ['c', 'd']**

**['1', '2'] \* 3**

**a\_string = "hohoho"**

**an\_int = 12345**

**a\_list = ["hello", 4, a\_string, 5.6, "a", an\_int, True]**

**a\_list + ['additional string']**

**# check if a\_list has changed**

**a\_list**

##### Membership checking (using operator in)

The operator in allows us to check if an object is an element of a list.

###### Example [CS-3] - Checking for list membership (run interactively)

**theList = ['abc', 3, True, '']**

**'' in theList**

**3 in theList**

**4 in theList**

##### Accessing parts of a list by index (using operator [])

As with strings, square brackets can be used to access a single element of a list or a group of contiguous elements. The way indices are used with the square brackets is exactly the same as for accessing substrings. Positive and negative index values can both be used.

###### Example [CS-4] - Accessing elements of a list (run interactively)

**laugh = "hohoho"**

**letters = ["a", "b", 'c']**

**a\_list = ["hello", 4, letters, 5.6, "a", laugh, True]**

**a\_list**

**a\_list[0]**

**a\_list[2:5]**

**a\_list[5:]**

**a\_list[:-5]**

**# accessing element in sublist**

**a\_list[2][1]**

**# accessing substring in string element**

**a\_list[5][:2]**

##### Comparison

###### Equality operators

These can be used between any two lists. For two lists to be equal, they need to be of the same length and their elements in the same position must be of the same type and have the same value.

Example [CS-5] - Equality of lists (run interactively)

**['three'] == [3]**

**['3', 4] == ['3', 4]**

**[2] != [3]**

**['one', 'two', 'three'] == ['one', 'two']**

###### Relational operators

Two lists can be compared with relational operators if their elements in the same position are individually comparable. Comparisons are carried out left to right i.e. the relation between an element pair carries more weight if the elements are at a lower index.

Example [CS-6] - Relational operations on lists (run interactively)

**['3', 4] < ['3', 4] # False**

**['2', 4] < ['3', 3] # True**

**['2', 4] < [3, 3] # Error! ('<' not supported between instances of 'str' and 'int')**

#### Operations that change the list object

With square brackets (**the [] operator**) we can:

###### Example [CS-7] - assign a new object to a list element Setting individual element values of a list

**myList = ["hello", "hi", 5, "how are you"]**

**myList**

**myList[2] = "bonjour"**

**myList**

###### Example [CS-8] - replace any contiguous portion of a list with another list Replacing a sublist

**myList = ["hello", "hi", 5, "how are you"]**

**myList**

**myList[2:3] = ["bonjour", "hola"]**

**myList**

**myList[-1:] = ["good day"]**

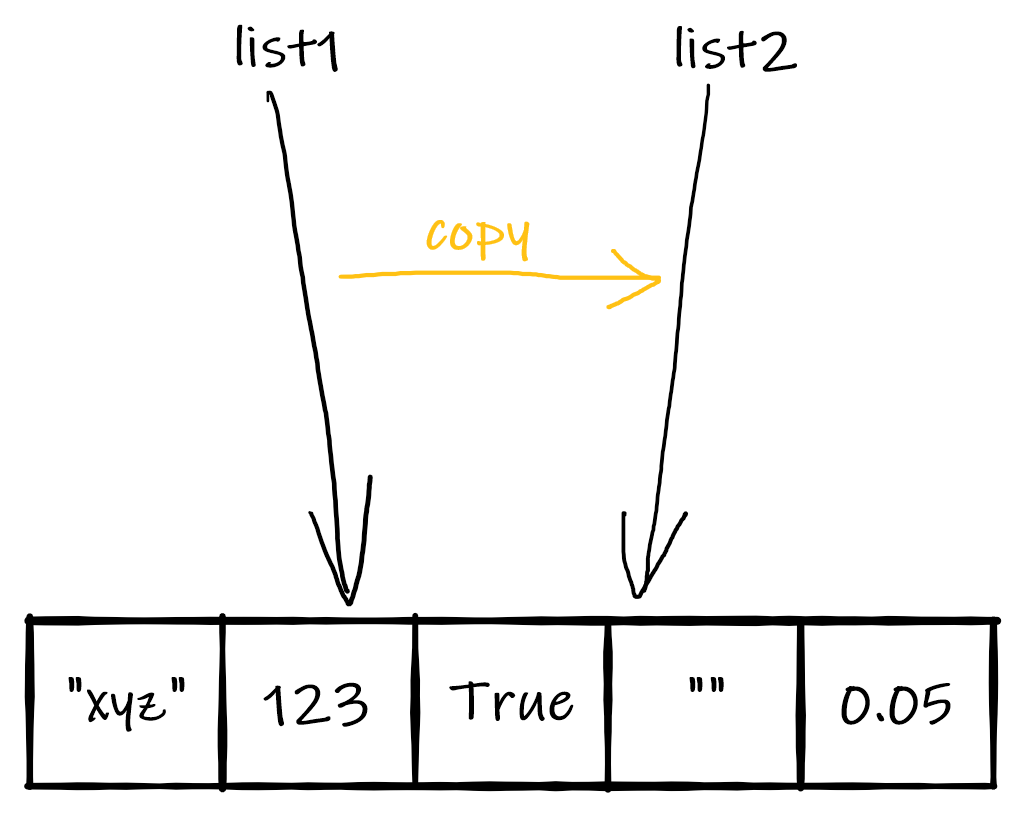
**myList**

#### Copying lists

There are several levels of copying that can be carried out with lists.

##### Creating a new name for a list vs. copying

###### New name

Giving an existing list a new name (list2 = list1) 

Assigning a list object to a new name results in the creation of a new name for the list. The new name points at the exact same list in memory that the assigned object is at. What is copied is only the connection (pointer) from name to object. It is a **copy of a reference**.

Example [CS-9] - Create a new name for a list (run interactively)

**list1 = ['abc', 123, 1.23, True]**

**list2 = list1**

**list1**

**list2**

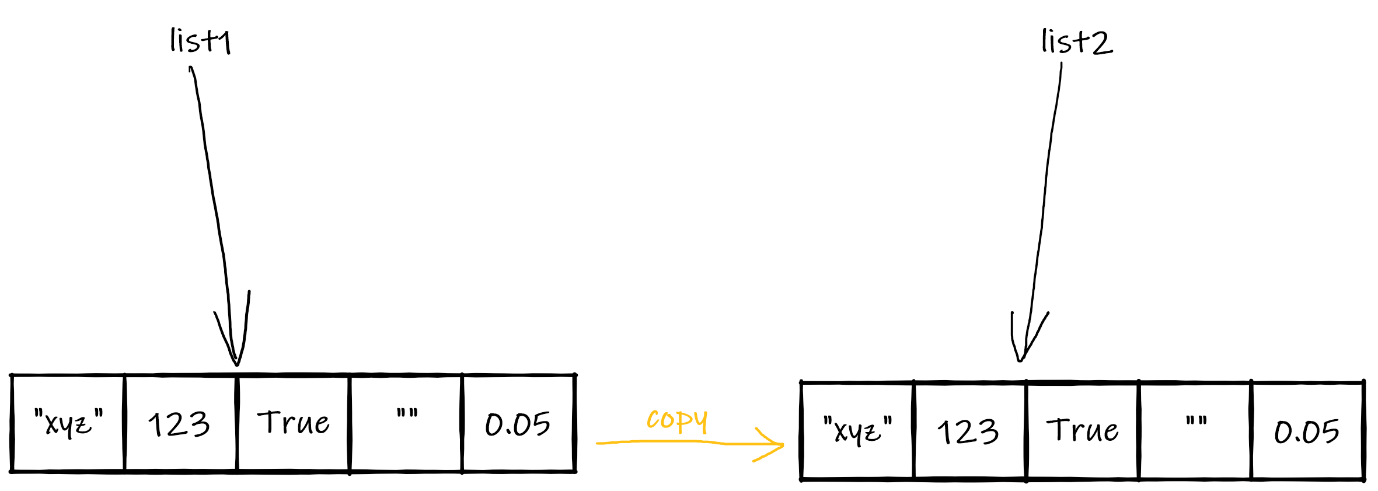
**# check it is the same list**

**list1[1] = 456**

**list1**

**list2**

###### **Copy**

**Copy (list2 = list1.copy() or list2 = list1[:])** 

A copy of a list results in the creation of a new name for the list and a copy of the content in a new memory location. The new name points to the copy of the content. It is a **copy of a value**.

Example [CS-10] - List copy (run interactively)

**list1 = ['abc', 123, 1.23, True]**

**list2 = list1[:]**

**list3 = list1.copy()**

**# check copies are different lists**

**list2.append('newEl')**

**list3.remove('abc')**

**list1**

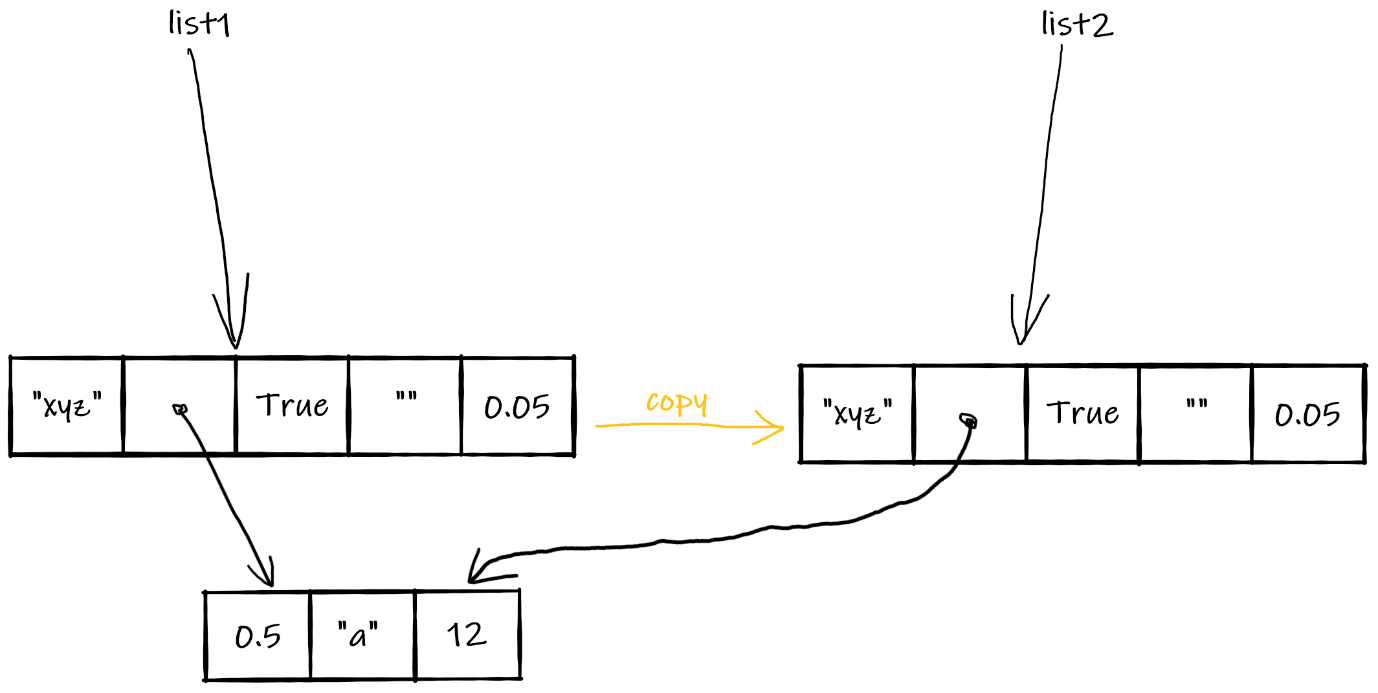
**list2**

**list3**

###### Shallow copy vs. deep copy

Copying a list with nested lists requires a special operation i.e. a **deep copy**. The copy discussed above is known as a **shallow copy**.

Shallow copy

Copy of list containing lists (list2 = list1.copy() or list2 = list1[:]) 

The figure *Shallow copy* shows what happens when the function copy() is called on (or equivalent operator [:] is applied to) a list with nested lists.

The only object copied is the top-level list. For nested lists, only the reference (pointer) is copied.

*Example [CS-11] - List copy (run interactively)*

**list1 = ['abc', [1, 2, 3], 1.23, True]**

**list2 = list1[:]**

**# see that the lists are the same**

**list1**

**list2**

**# change original list top level**

**list1[2] = 4.567**

**# change original list nested level**

**list1[1][1] = 222**

**# see list1**

**list1**

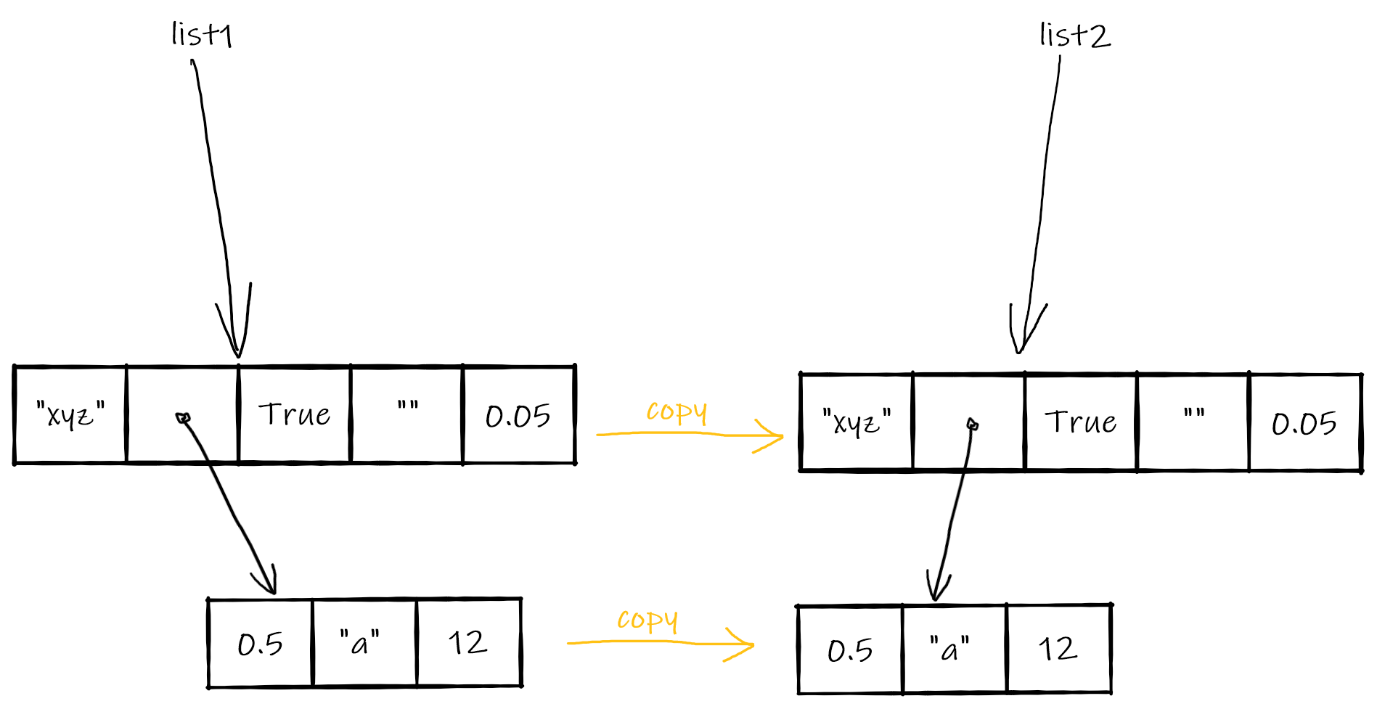
**# see list 2**

**# top level is not modified**

**# nested is modified (because shared)**

**list2**

Deep copy

Deep copy of list containing lists (list2 = copy.deepcopy(list1)) 

When a deep copy is made, the whole hierarchy of nested lists is copied, to any level of depth. For this, the function copy.deepcopy().

*Example [CS-12]--- List copy (run interactively)*

**# import the copy module**

**import copy**

**# define list1 and create a deep copy**

**list1 = ['abc', [1, 2, 3], 1.23, True]**

**list2 = list1.deepcopy()**

**# see that the lists are the same**

**list1**

**list2**

**# change original list top level**

**list1[2] = 4.567**

**# change original list nested level**

**list1[1][1] = 222**

**# see list1**

**list1**

**# see list 2 (no modifications)**

**list2**

#### Creating Lists from strings

##### Casting a string to list

Casting a string to a list returns a list with one string element for each character in the original string.

Example [CS-13]--- Casting strings to lists (run interactively

**myName = "Jane Brown"**

**list(myName)**

##### Splitting a string

With the split() method we have more control over how the string is split into list elements, in that a separator can be specified. When the separator is not specified, the split creates substrings of non-whitespace characters that were separated by whitespace characters in the original string.

Example [CS-14]--- The split method (run interactively)

**myName = "Jane Brown"**

**myName.split()**

**wordsStr = "last, first, penultimate, second, unspecified"**

**wordsList = wordsStr.split()**

**wordsList**

**type(wordsList)**

**# use a separator**

**wordsStr.split(sep=", ")**

#### Creating strings from lists

##### Casting

Casting from list to string will not produce the result we generally want. The returned string is just the list, printed out.

###### *Example [CS-15]---* Casting list to string (run interactively)

**str(["a", "b", "c"])**

##### Joining elements into a string

The string type provides a method [**join()**](https://docs.python.org/3/library/stdtypes.html#str.join) for joining elements of an iterable object (which a list is) into a string.

The syntax for calling the method is **<separator string object>.join(<list object>),** i.e. the method is called on a string object that should act as separator in the resulting string.

###### *Example [CS-16]---* Joining list elements into a string (run interactively)

**' '.join(["a", "b", "c"])**

**''.join(["A", "n", "n"])**

**separator = ", "**

**list = ["python", "javascript", "java"]**

**separator.join(list)**

### Transforming lists

#### Lambda Expressions

##### A very short introduction to lambda expressions

A lambda expression

* is an expression that defines a function i.e. a reusable piece of functionality
* returns (evaluates to) an **anonymous** function i.e. a function without a name
* can be called as a function
* can be assigned to a variable and thus get a name by which it can be called
* can be passed in as an argument to another function and called by the parameter name
* takes the form

**lambda <parameter list> : <return value created from parameter>**

* takes arguments defined by the parameter list between lambda and the colon
* returns the value to which the expression right of the colon evaluates
* can use variables defined outside of it

##### Examples [CS-17] - Examples of lambda expressions and their use (run interactively)

**# define a lambda expression**

**# that increases a number by 3**

**# and assign it to f (name it f)**

**f = lambda x : x + 3**

**# run it**

**f(4)**

**f(2.3)**

**# we don't have to name a lambda**

**# expression but can call it**

**# directly (it must be placed**

**# in parentheses)**

**(lambda x : x + 3)(4)**

**(lambda x : x + 3)(2)**

**# define and use a lambda**

**# expression with two parameters**

**(lambda x, y : x \* y)(3, 4)**

**(lambda x, y : x \* y)(23, 2)**

**# define a lambda expression**

**# that checks if a number is**

**# greater than 3 (the return**

**# type is boolean)**

**g = lambda x : x > 3**

**# run it**

**f(3)**

**f(3.1)**

##### Examples [CS-18] - Examples of lambda expressions that use variables external to them

**# some names**

**names = ["ben", "con", "ann", "dan"]**

**# define a lambda expression that returns the**

**# index in the list 'names' for the given string value**

**get\_index\_for\_name = lambda nm : names.index(nm)**

**# use it**

**get\_index\_for\_name('ann')**

**get\_index\_for\_name('ben')**

**# some scores**

**scores = [23, 67, 34, 12]**

**# define a lambda expression that returns the**

**# value in the list 'scores' corresponding by position**

**# to the given value in the list 'names'**

**get\_score\_for\_person = lambda nm : scores[names.index(nm)]**

**get\_score\_for\_person("ben")**

**get\_score\_for\_person("dan")**

**# sort the name list by scores, using**

**# sorted(), which does not modify the list;**

**# we need the original list intact as**

**# it is being used in the lambda expression**

**sorted(names, key=get\_score\_for\_person)**

**# or**

**sorted(names, key=lambda n : scores[names.index(n)]**

List index() Method Syntax

**list\_name.index(element, start, end)**

#### Sorting Lists

Lists can be sorted using **the** [**sort()**](https://docs.python.org/3/library/stdtypes.html#list.sort) **method**, which modifies the list **in place**. The **function** [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) **has the same functionality but does not modify the list, creating a copy of it to sort and return.**

For a list to be sortable, either

1. the values of all its elements need to be comparable

##### Example [CS-19]---Sorting a list without a key (run interactively)

**names = ['ben', 'con', 'don', 'ann']**

**names**

**names.sort()**

**# names has changed in place**

**names**

**# now try using sorted() to preserve the original list**

**games = ['scrabble', 'ludo', 'monopoly', 'carcasonne']**

**games**

**sorted(games)**

**# check games again (should not have changed)**

**games**

**# try to sort a list with types that cannot be compared**

**mixed\_list = ['abcd', 34, '', 'aaa', 0, 4, 0.0]**

**mixed\_list.sort() # error!**

**# however if all types can be compared:**

**another\_mixed\_list = [2, True, 0.2]**

**another\_mixed\_list.sort**

**# True compared as 1 and the sort works  
another\_mixed\_list**

b) the key argument is a function (either named or [lambda expression](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_6_lambda.html)) that converts all the elements to comparable values

##### Example [CS-20] - ---Sorting a list with a key (run interactively)

**# use the bool() casting function as key**

**#-------------------**

**mixed\_list = ['abcd', 34, '', 'aaa', 0, 4, 0.0]**

**mixed\_list.sort() # error**

**# pass key to convert to boolean**

**mixed\_list.sort(key=bool)**

**# elements that equate to False are listed first**

**mixed\_list**

**# use the len() function as key**

**#-------------------**

**word\_list = ['hello', 'hi', 'how are you']**

**word\_list**

**word\_list.sort(key=len)**

**# words are sorted by length**

**word\_list**

**# use a lambda expression as key**

**#-------------------**

**# some names**

**names = ["ben", "Con", "ann", "Dan"]**

**# try to sort them**

**names.sort()**

**# order is not alphabetical because**

**# because some names are capitalised;**

**# we use a lambda expression to**

**# convert to all-lower before sorting**

**names.sort(key=lambda n : n.lower())**

(lambda (some\_variable): what ever you need to do, (mapped to whatever list you want to apply itto)

**Lambda is an expression that generates a new function to be called later – as it is an expression it can be used in places where def cannot i.e. lists and dictionary litterals… (p505)**

#### Map and filter

While sorting results in a list with the same elements but possibly in a different order, the [**map()**](https://docs.python.org/3/library/functions.html#map) function maps each element of the list to create a list of new elements, while the [**filter()**](https://docs.python.org/3/library/functions.html#filter) function applies a filter to each element of the list to create a possibly shorter list.

* The syntax of the map function is **map(<mapping function>, <list to be mapped>).** The function returns an iterable object, which means that we need to cast it to a list to get a list with the elements resulting from the mapping. Thus the full expression for mapping a list to a list is so **list(map(<mapping function>, <list to be mapped>)).**

##### Example [CS-21] - Mapping a list to a new list

**names = ['jim', 'john', 'jane', 'jen']**

**# map to name lengths**

**list(map(len, names))**

**# map to capitalised names**

**list(map(lambda s : s.upper(), names))**

* The map function can work on several input lists in parallel. They all have to be the same length.

##### Example [CS-22] - Mapping two lists to a new list

**names = ['jim', 'john', 'jane', 'jen']**

**surnames = ['lock', 'smith', 'stone', 'mason']**

**# join name and surname, capitalising both**

**list(map(lambda n, s : n.capitalize() + " " + s.capitalize(), names, surnames))**

* The syntax of the filter function is **map(<filter function>, <list to be filtered>).** The filter function must return a boolean value. Similarly to the map function, the result of the filter function must be explicitly transformed to a list.

##### Example [CS-23] - Filtering a list

**names = ['jim', 'john', 'jane', 'jen']**

**# find the names that contain an 'n'**

**list(filter(lambda n : n.find('n') >= 0, names))**

#### List comprehensions

List comprehensions are a special construct that combines mapping and filtering of lists. In most cases, it is easier to read than the map/filter combination. The syntax of a list comprehension is

**[** expression1 **if** condition1 **else** expression2 **for** list\_element **in** list **if** condition2 **]**

* The black text in the syntax specification constitutes the **fixed parts** of the comprehension, including keywords and brackets.
* The statement for list\_element in list specifies the list with expression list and a variable to store the current element (list\_element) during iteration over the list , much like in a usual for-in loop.
* expression1 is the mapping expression, which transforms each list element to an element of the new list (it corresponds to the function or lambda expression passed to the map function).
* The part with the blue background is the filter, with condition2 as the filtering condition, which must return a boolean value. This part is optional.
* The part with the green background is a construct that allows conditional mapping: expression1 is used if condition1 is true, while expression2 is used if not. This part is also optional.

##### Example [CS-24] - Examples of list comprehensions

**names = ['jim', 'jill', 'john', 'jane', 'jen', 'jack']**

**# examples with no filter or conditional mapping**

**# -----------------------------------------------**

**# map to string length**

**[ len(n) for n in names ]**

**# map to names in all-capitals**

**[ n.upper() for n in names ]**

**# examples with filter only**

**# --------------------------**

**# map to string length, including only names containing an 'e'**

**[ len(n) for n in names if 'e' in n ]**

**# map to names in all-capitals, including only names longer than 3**

**[ n.upper() for n in names if len(n) > 3 ]**

**# examples with conditional mapping only**

**# ---------------------------------------**

**# map to all-capitals if name length is less than 4, otherwise don't change**

**[ n.upper() if len(n) < 4 else n for n in names ]**

**# map to length+1 if name contains an 'e', otherwise map to length-1**

**[ len(n) + 1 if 'e' in n else len(n) - 1 for n in names ]**

**# examples including all parts of the construct**

**# ----------------------------------------------**

**# map to all-capitals if name length is less than 4, otherwise don't change,**

**# including only names with an 'e' (i.e. only if it has an e in the name)**

**[ n.upper() if len(n) < 4 else n for n in names if 'e' in n ]**

**# map to length+1 if name contains an 'e', otherwise map to length-1,**

**# including only names longer than 3 (only if**

**[ len(n) + 1 if 'e' in n else len(n) - 1 for n in names if len(n) > 3]**

Comprehensions are easier to write and to the trained eye are much easier to read. There are some scenarios where map/filter is still more concise: when the mapping function is built in (e.g. len or int) and when multiple lists need to be transformed.

In the list comprehension in the example below, the **zip** function needs to be **called to combine the names and surnames into tuples (n, s),** which are then transformed. This is a sort of a hack, since it creates an intermediate data structure that is needed only for processing.

**Maps vs comprehension(neither is correct/incorrect, whatever you are used to )**

* **Comprehension can only work on one list, so need to use zip function**
* Map>> use as Jelena used to them
* Filter >>
* Map+ filter is mapping function is already in exiostence
  + i.e. len function a spre-defined
* mapswill work with more than one list…

##### Example [CS-25] - Example of list comprehension using two input lists

**names = ['jim', 'jill', 'john', 'jane', 'jen', 'jack']**

**surnames = ['lock', 'smith', 'stone', 'mason', 'bell', 'ringer' ]**

**# join name and surname, capitalising both, using map**

**#**

**list(map(lambda n, s : n.capitalize() + " " + s.capitalize(), names, surnames))**

**# the same transformation using a list comprehension**

**[ n.capitalize() + " " + s.capitalize() for (n, s) in zip(names, surnames) ]**

### Tuples and enumerate

In the section on list comprehensions, we have seen a structure called the **tuple**. [Tuples](https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences) are sequences, like lists, but are immutable. They are used in contexts where multiple values are needed instead of one value. Tuple literals can be defined either as a comma-separated list or a comma separated list in parentheses.

Tuples Is a list of variables – for cases where we need more than one variable, can contain any number of elements, really very much like lists but **not modifiable – are immutable like strings**

#### Example [CS-26] - Defining tuples

**t1 = 1, 2, 3**

**t2 = (1, 2, 3)**

**t3 = 3, 4, 5**

**t1 == t2 # expected output: true**

**t1 == t2 # expected output: false**

Python has a built-in function called **enumerate,** which returns index-value tuples of a sequence. This is very useful where the **indices of list elements are needed for processin**g. Like map and filter, this function's output is not a list but can be converted to one.

#### Example [CS-27] - Enumerate function

**names = ['jim', 'john', 'jane', 'jen']**

**# create an enumerated list**

**list(enumerate(names))**

**# get a list of strings preceded by 1-based indices**

**[ str(x[0] + 1) + " " + x[1] for x in enumerate(names) ]**

**# or**

**list(str(x[0] + 1) + " " + x[1] for x in enumerate(names))**

### Picking a random element from a list

Many programs will require the generation of some sort of random information. Usually, a random number is generated in a set range.

Python has a function that allows picking a random item from a list directly, **random.choice().**

#### Example [CS-28]--- Picking a random element (run interactively)

**aList = ['hello', 4, ['a', 'b', 'c'], 5.6, 'a', 'hohoho', True]**

**aList**

**import random**

**random.choice(aList)**

**random.choice(aList)**

**random.choice(aList)**

**random.choice(aList)**

**random.choice(aList)**

## Simple program design steps (roughly!)

1. Identify inputs and outputs
2. Identify the order of inputs and outputs
3. Identify which inputs affect which outputs
4. Design the flow of control
5. Identify repetitions and variables
6. If necessary, go back to step 4
7. Fill the programmatic details
8. After implementation and testing, fix any imperfections

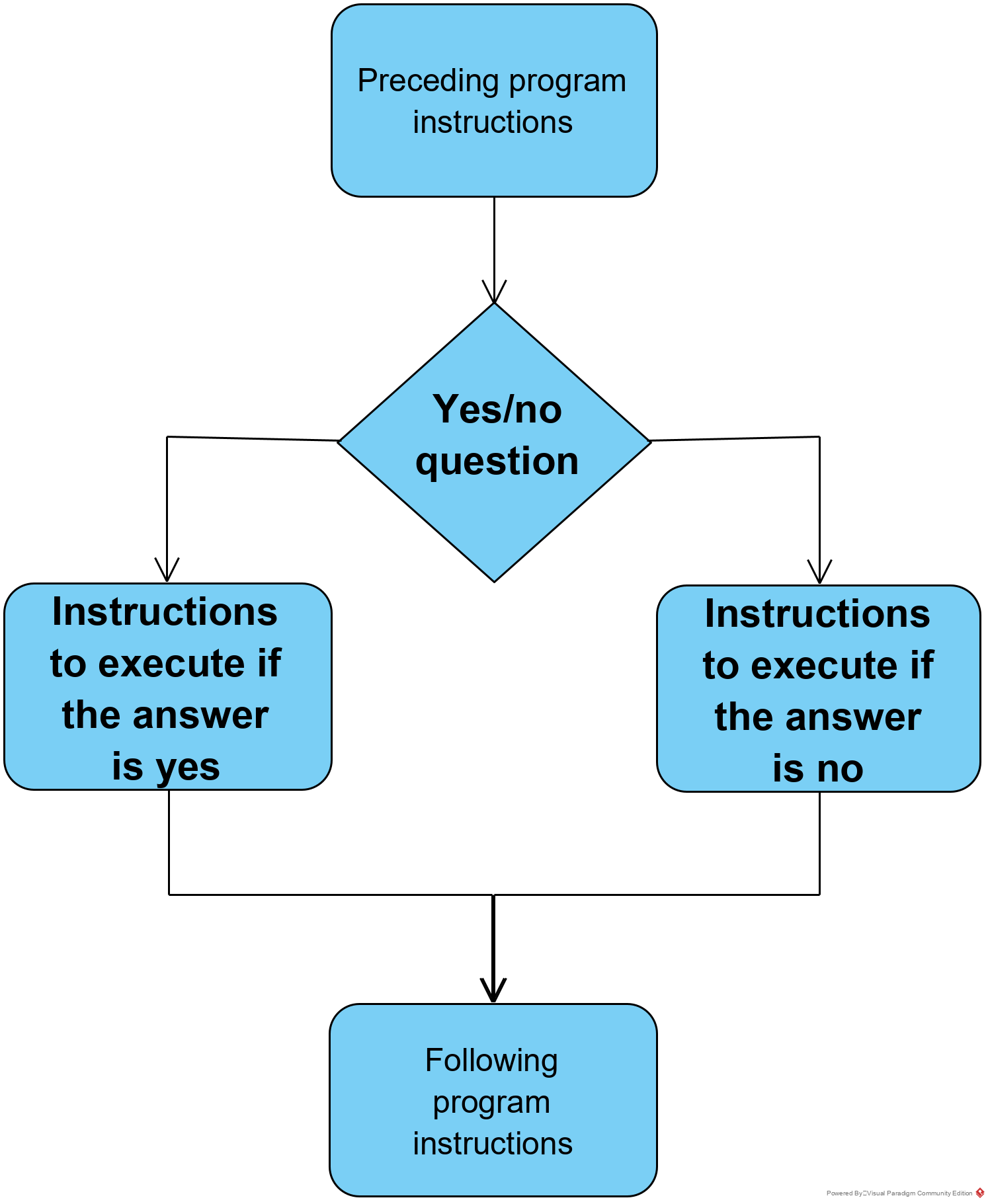
# Week 3

## Conditionals

To conditionally execute code need a conditional statement… condition the flow…

### What is a conditional statement

Exists in all imperative programming languages.. and functional progamming langs.. can’t program without it

The flowchart of basic conditional statement in a computer program – basic chart that applies to all langs not just python

The conditional statement is one of the three essential ingredients needed for expressing algorithms:

* Assignments (assign to variables/lists etc)
* **conditional statements**
* loops

### The basic conditional statement in Python

##### Example [CS-1] - Example of a basic conditional statement in Python

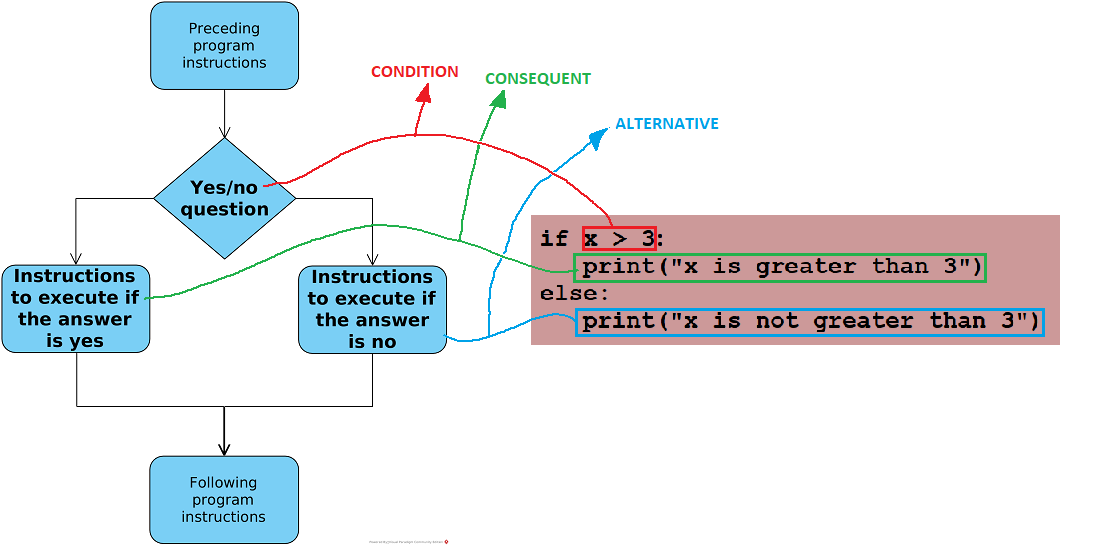
Indentation ism meaningful – must be indented by same amount in python. Is binary (doesn’t matter how much you indent -indentation is meaningful in python) is Binary.. is either met or it isn’t…)

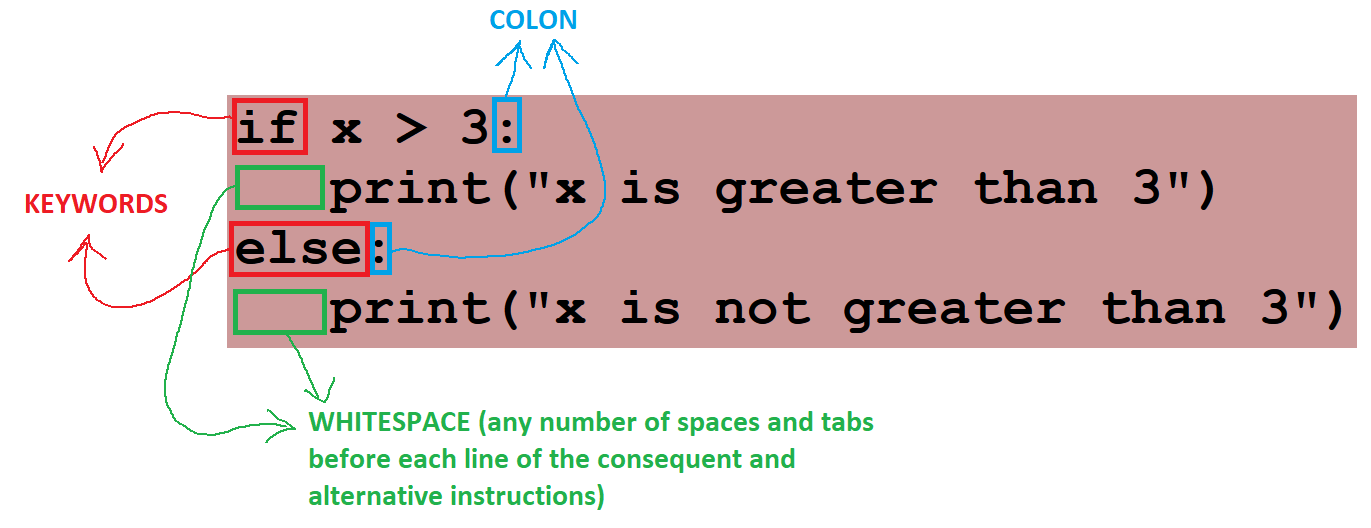
**if x > 3:**

**print("x is greater than 3")**

**else:**

**print("x is not greater than 3")**

Relating a Python conditional statement to the generic flowchartThe syntax of a Python conditional statement

Answer **must translate as a Boolean** i,e. true or false (condition must be yes or no)

* + The **condition** must be a yes/no question, which in programming language terms means that it must **evaluate to a boolean**.
  + Both the **consequent** and the **alternative** can be any sequence of programming instructions.

#### In-class exercise [CE-A]

Use the basic conditional statement example above in a runnable script that checks a number input by the user and tells the user if the number is greater than 3.

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

#### In-class exercise [CE-B]

What will the following program print?

**if True:**

**print("It's true")**

**else:**

**print("Not a chance")**

### Conditional statement variations

#### Without an alternative ('if' statement)

1. The else keyword and the alternative instructions listed below it can be left out entirely

###### Example [CS-2] - Example of an 'if' statement in Python (run interactively)

**if x > 3:**

**print("x is greater than 3")**

* In this case the alternative consists of no instructions i.e. **nothing is done if the condition is not fulfilled.**
* By writing this kind of conditional statement, we are implicitly stating the alternative and should always keep that in mind. Not considering it can lead to errors.

##### Example [CS-3] - A script with a problem caused by an omitted branch

**x = 4**

**if x == 3:**

**y = x \* 2;**

**print(y)**

#### In-class exercise [CE-C]

Write a script to try out the code above. What happens when you run it?

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

#### With more than two branches

* + Conditions evaluate to booleans and can have only two outcomes, hence the only way to split the flow into more than two branches is by using **nested conditional statements** that split the already existing branches.

##### Example [CS-4] - Example of creating multiple branches of execution

**if x > 3:**

**if x > 10:**

**print("x is greater than 10")**

**else:**

**print("x is greater than 3 but not greater than 10")**

**else:**

**if x > 0:**

**print("x is 1, 2 or 3")**

**else:**

**print("x is not a positive number")**

#### The elif keyword

* When a nested conditional statement is inside the *alternative* part of the conditional statement i.e. in the block that follows the else keyword, the else and if keywords can be collapsed into the keyword **elif**. With the use of elif the code snippet above turns into the following.

###### Example [CS-5] - Example of using the elif keyword

**if x > 3:**

**if x > 10:**

**print("x is greater than 10")**

**else:**

**print("x is greater than 3 but not greater than 10")**

**elif x > 0:**

**print("x is 1, 2 or 3")**

**else:**

**print("x is not a positive number")**

The logic of code snippets [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-multi-branch) and [[CS-5]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-elif-intro) is identical.

* + In order to 'get rid of' the first nested conditional statement in code snippet [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-multi-branch), we would have to **change the order** in which the conditions are checked.

By reordering the checking of the conditions and using the elif keyword, write a code snippet logically equivalent to [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-multi-branch) that does not have any nested conditional statements.

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

### Covering all bases

When programming conditional statements it is important to **stay in control** by accounting for both branches of every condition, implicit or explicit.

#### Implicit alternatives

* + An alternative branch may be implicit with good reason, but deciding this must be part of the programming process.
  + For example, I may want to issue a warning inside a stock-keeping application if the stock of a product is very low. If the stock is not low I *do not want the program to do anything*. This is a valid candidate for a conditional statement without an alternative branch. I am sure that nothing needs to be done if the condition is not met i.e. if there is enough stock.
  + Cases where it makes sense to not have the “else” as don’t’ want it to do anything

##### Example [CS-6 ] Example of a conditional statement validly without an alternative

**stock = 0 /\* calculated in some way \*/** *(this is java format for comments)*

**if stock < 2:**

**print("WARNING: Stock is very low. Put in an order with supplier!")**

**/\* no else needed \*/**

* + Nested conditional statements can make it harder to see 'un-covered' branches of execution. Consider the following example.

##### Example [CS-7 ] - Example of a missing alternative

**age = int(input("Please enter your age:"))**

**hasProvisional = input("Do you have a provisional licence (y/n):")**

**if age >= 17:**

**if hasProvisional == "y":**

**print("You are eligible for a driving licence.")**

**else:**

**print("You are not eligible for a driving licence.")**

Spot the problem in code snippet [[CS-7]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-if-no-alt-incorr) and rewrite it so that it works correctly. Start by using a flowchart to identify the required algorithm.

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

#### Hidden alternatives (don’t use if if if if… )

* + For all conditions, the condition and it's negative can be programmed into separate conditional statements. While this approach can be logically correct, it is
    - inefficient (uses more computing power)
    - harder to write
    - harder to understand and maintain
  + Look at the following example where we want the program to do one thing when x is equal to 3 and another when it is not equal to 3. The two conditional statements 'cover all bases' but inefficiently and 'messily'.

###### Example [CS-8] - Example of hidden alternatives

**print("Guess my lucky number (it's an integer): ", end="")**

**x = int(input())**

**if x == 3:**

**print("You have guessed my lucky number!")**

**if x <> 3:**

**print("Sorry, that's not my lucky number :-(")**

##### In-class exercise [CE-F]

Rewrite code snippet [[CS-8]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_7_conditionals.html#py-if-no-alt-bad) so that it takes advantage of the Python conditional statement's built in alternative handling.

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

### Complex conditions

1. We use the logical operators **AND**, **OR** and **NOT** (keywords **and**, **or** and**not**) to combine conditions into complex conditions.
2. Depending on the logic, complex conditions can be used to 'flatten', partially or fully, a nested conditional statement structure.

#### In-class exercise [CE-G]

Rewrite the driving licence example to use a complex condition instead of a nested conditional statement.

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

### Exercises

#### In-class exercise [CE-H]

Write a Python script for Icy Icecream Parlour's automated cash register. It should:

* + Communicate appropriately with the customer, printing prompts when needed
  + Read in the number of icecream scoops that the customer wants to buy
  + Calculate the amount due, based on a scoop price of €1.20
  + Ask the customer if they are over 65 and apply a discount of 20% to the price if they are
  + Print the amount due

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

#### In-class exercise [CE-H]

Write a Python script for Cringy Cinema film listing machine. The script should not contain any nested conditional statements. It should:

* + Have three films stored as string variables: a G-rated, a 15-rated and an 18-rated film
  + Communicate appropriately with the customer, printing prompts when needed
  + Read in the customer's age
  + Print a list of suitable films

[EXAMPLE SOLUTION 1](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/conditionals/cond_cinema.py)

[EXAMPLE SOLUTION 2](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/conditionals/cond_cinema2.py)

**Note on the example solutions**

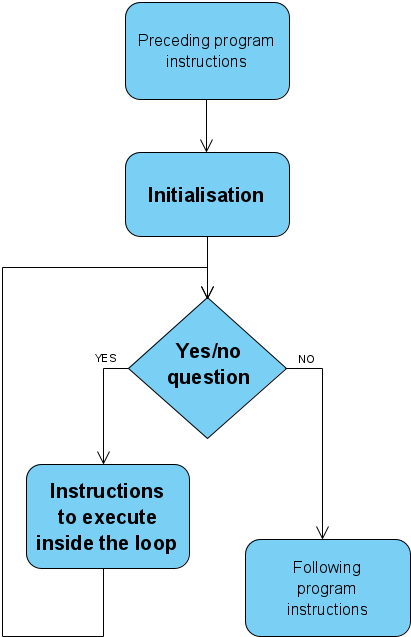
Both solutions are logically correct. The first one uses one conditional decision tree to decide which list of films to print. The second one uses two conditional statement trees, one for the over-15 and another for the over-18 film.

The first example is a larger program i.e. would take up more space in memory. The other example (owing to the two decision trees) is slower. While the difference in size (three function calls) and speed (ratio 5:6 for conditional checks) are negligible in these small examples, in real-life projects design decisions of this kind are made as part of code optimisation.

*In class the main goal is to design****correct****solutions. If a particular form (e.g. an elif 'flat' decision tree) is required, this will be stated in the assignment specification but otherwise any correct solution is satisfactory.*

## While loops

### What is a loop in programming?

The flowchart of a generic loop in a computer program

The loop is one of the three essential ingredients needed for expressing algorithms:

* + assignments
  + conditional statements
  + loops

A loop is **used when the same instructions need to be iterated (repeated several times**):

* + to improve the development process (speed, readability and maintainability)
  + to optimise the program (smaller code size)
  + to make use of the processing power of a computer (i.e. get things done on a scale that humans cannot achieve)
  + to allow the program to be written at all:
    - the number of iterations may not be known in advance if it depends on environment communication or is random
    - if the program represents a 'server' (runs infinitely)

### The while loop in Python

The main function of a loop is to **repeat some group of instructions over and over again**. The loop in code snippet [[CS-1]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-inf) does exactly that: it prints out the number 1 over and over again. The condition is number == 1 and in this context always true. The call to time.sleep() has been added to space the iterations by 1 second, to avoid looping at a speed that hogs the CPU and freezes the computer running it. Once running, this little program will have to be stopped externally (**Ctrl-C** on the command line or stop button in PyCharm).

##### Example [CS-1] - Example of an infinite while loop in Python

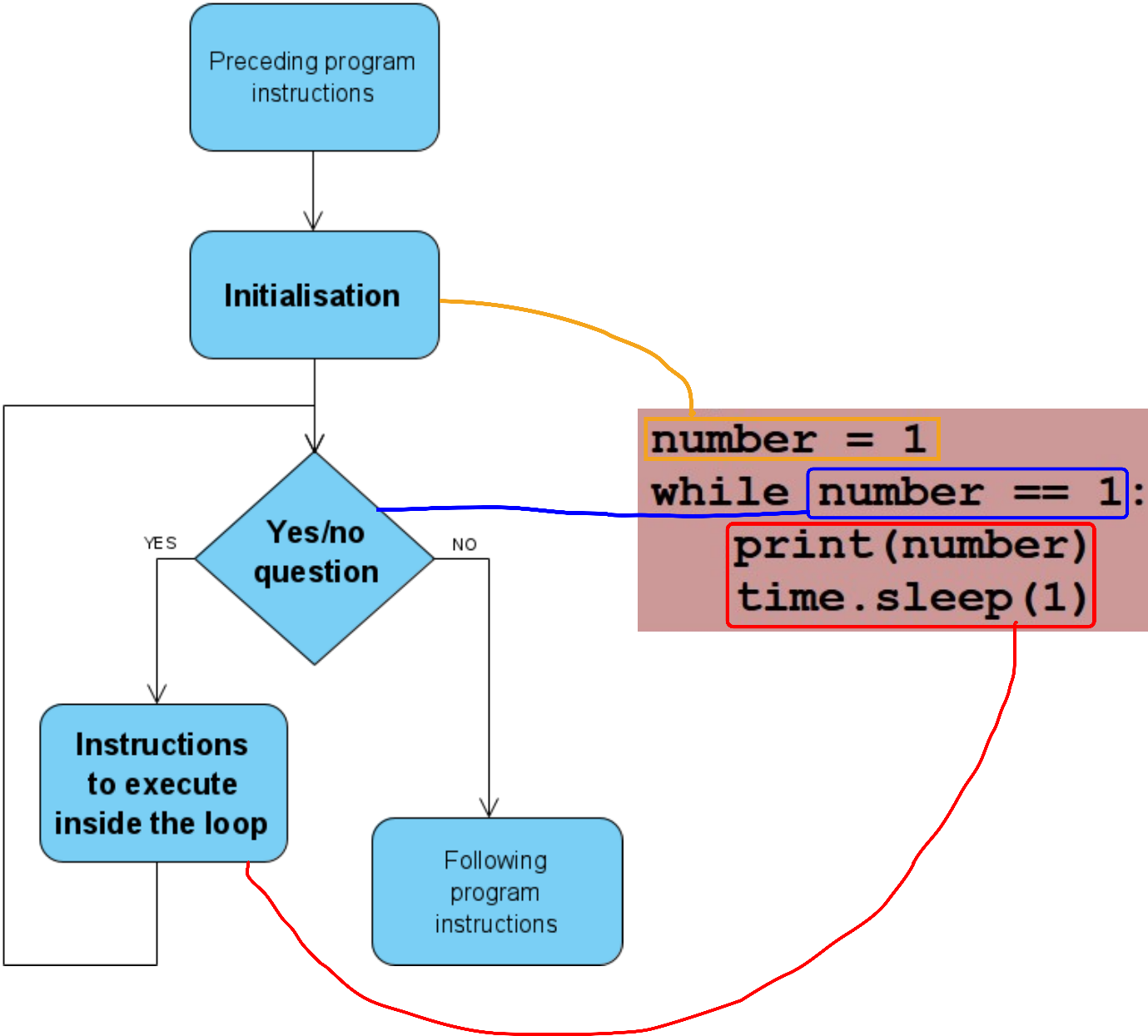
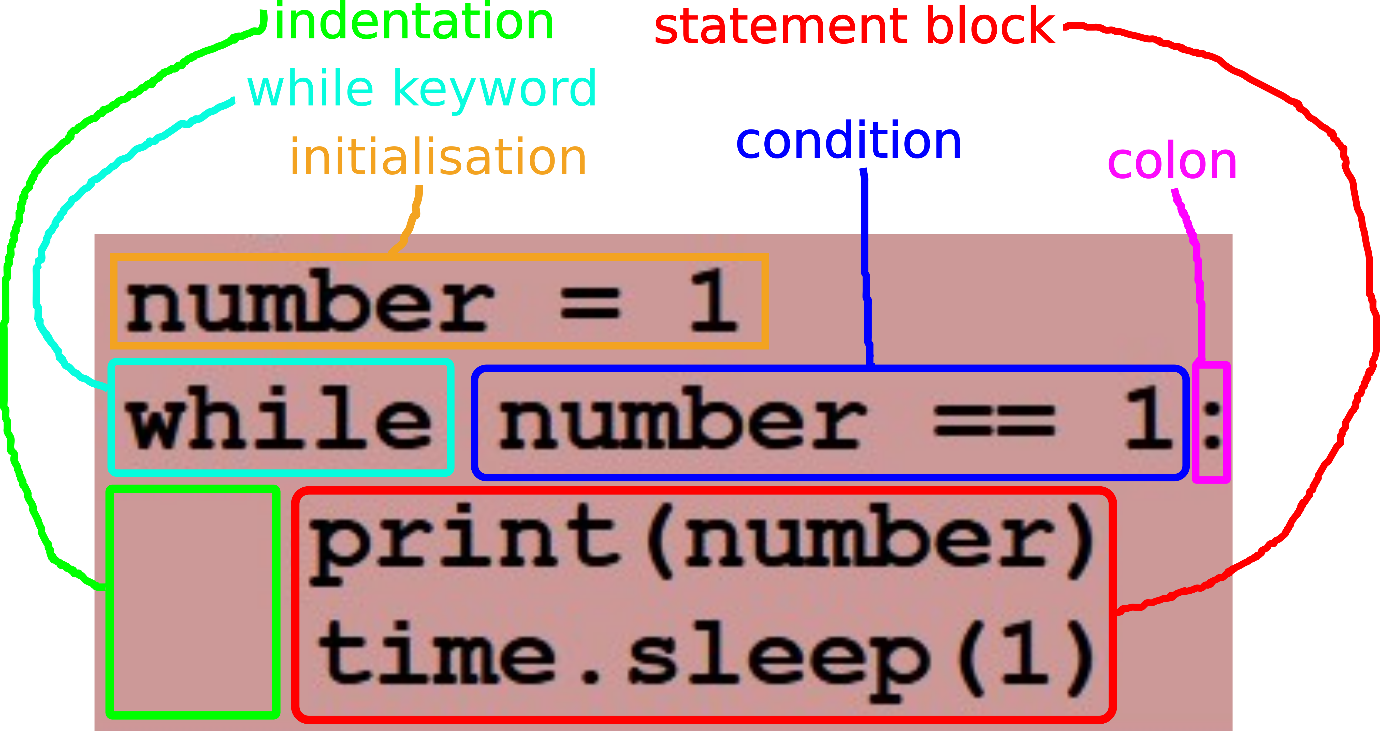
**import time**

**number = 1**

**while number == 1:**

**print(number)**

**time.sleep(1)**

Relating a Python while loop to the generic flowchartThe syntax of a Python while loop

### Exiting the loop

Most of the time, loops do not run infinitely and are written so that the conditional expression evaluates to False after a finite number of iterations. This can happen in different ways.

* + The variable checked in the conditional expression may be one that assumes values from a **fully determined sequence**, eventually reaching a value that makes the expression False. In this case the number of iterations is known in advance.

##### Example [CS-2] - While loop with the number of iterations fully determined at the time of coding

**number = 1**

**while number < 11:**

**print(1)**

**number = number + 1**

#### In-class exercise[CE-A]

Write a Python script that produces the output below. Use a while loop to print the lines between the first and the last one.

**Wherever repetition occurs that’s a candidate for a loop**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* \***

**\* \***

**\* \***

**\* \***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\***

[EXAMPLE SOLUTION](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/x4_print_rect.py)

#### In-class exercise [CE-B]

Write a loop that prints 10 random numbers between 1 and 11 (inclusive).

[EXAMPLE SOLUTION](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/x10_print_rand.py)

Often, the number of iterations cannot be determined at the time of coding but **at run-time is known before the loop is executed**. This is the case in code snippet [[CS-3]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-rt). Notice that if the user enters 1, the statement block of the loop will never be executed (0 iterations).

##### Example [CS-3] - A while loop with iterations fully determined at run-time

**iterCount = input("Please enter a positive whole number: ")**

**number = 1**

**while number <= iterCount:**

**print(1)**

**number = number + 1**

#### In-class exercise [CE-C]

Write a Python script that:

* generates a random number between 1 and 10 (inclusive) and stores it in a variable called iterationCount
* prints out the number 1 iterationCount times using a while loop

[EXAMPLE SOLUTION](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/xrand_print_1.py)

The variables checked in the conditional expression may, alternatively, depend on some **factor external to the program** and subsequently not be known to it before the loop runs. In this case the number of iterations cannot be known in advance. For example, a random number generator or user input, if used as the variable checked in the condition, would result in this scenario. The value obtained externally that halts the execution of the loop is often called the **sentinel** value.

##### Example [CS-4] - While loop with an unknown number of iterations owing to a random process

**import random**

**number = 1**

**while number != 11:**

**print(1)**

**number = random.randint(1, 11)**

#### In-class exercise [CE-D]

Rewrite code snippet [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-indet-rand) so that the loop statement block prints out the value of number rather than 1. The loop in [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-indet-rand) always prints out 1 at least once, because the random number is not generated or checked until the first iteration. Modify the script to generate a random number before entering the loop, resulting in no numbers printed if the first number 'drawn' is 11.

Test your script. It will take a few runs, but you will eventually get one where no numbers are printed out.

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

##### Example [CS-5] - While loop with exit conditioned on user input

**number = int(input("Please enter a number between 1 and 10 or 11 to stop: "))**

**while number != 11:**

**print(1)**

**number = int(input("Please enter a number between 1 and 10 or 11 to stop: "))**

##### Example [CS 6] - While loop with exit conditioned on data from a file (requires a text file called data.txt in the directory where it is running [[sample contents]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/data.txt))

**numList = []**

**f = open("data.txt", 'r') # opens file data.txt for reading**

**l = f.readline() # reads a line from the file**

**isNum = ''.join(l.strip().split('.', 1)).isnumeric()**

**while isNum:**

**numList += [ float(l) ]**

**l = f.readline()**

**isNum = ''.join(l.strip().split('.', 1)).isnumeric()**

**f.close() # closes the file**

**print(numList)**

### Iteration initialisation and choosing conditional variables

#### Desiging initial conditions for all iterations (not only the first one)

While initialisation is generally explicitly named as part of the standard while loop structure, variables (both conditional and other) must be set up in each iteration in preparation for the next one. The initial conditions of the loop are available to the first iteration as it is executed. The conditions created in the first iteration are available to the second iteration and so on.

For example, in code snippet [[CS-2]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-ct), the variable number must be 'set up' for each iteration:

* + for the first iteration, it is set to 1 (before the loop starts)
  + for subsequent iterations, the preparation consists of number being incremented

In this example number represents the iteration number and is not used in any way except as a conditional variable.

##### In-class exercise[CE-E]

Examine all the code snippets and in-class exercise solutions and identify all the set up that is performed for the loop iterations.

#### Choosing conditional variables

It is often a good idea to design the statement block (the body) of the while loop first, then identify conditional variables. Most of the time the conditional variables can be chosen from variables already used in the statement block. We will look at this process through some examples.

##### Example 1 - processing a predetermined sequence

Let's say we want to print out all the numbers from 1 to 10. We want the first iteration to print 1, the second to print 2 etc. (10 iterations in all).

* + First, let's **design the statement block**:
    - Because the number printed is changing from iteration to iteration, it needs to be a variable, e.g. n
    - Then we can easily write the main part of the statement block as print(n)
    - The last thing that is missing in the statement block is some code to set up n for the next iteration: n = n + 1
    - Also, n needs to be set up for the first iteration. The statement n = 1 placed before the loop achieves this.
  + Now we need to **design the while loop condition**. This takes a small bit of analysis. The variable n is moving upwards by one in each iteration and we want the looping to stop when n goes beyond 10. So n naturally lends itself as the basis for the condition, n < 11.

We can now write the short program. It is provided as code snippet [[CS-7]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-des-det-ct).

###### Example [CS 7] - A loop that prints out the numbers from 1 to 10

**n = 1**

**while n < 11:**

**print(n)**

**n = n + 1**

##### Example 2 - processing a sequence defined at run-time

Let's say we want to print out all the characters in a word input by the user, each character on a separate line followed by its UTF code in square brackets.

* + Again, let's design the statement block first:
    - The character changes between iterations so must be a variable, e.g. c
    - This makes it easy to write the printing code: **print("{0}[{1}]".format(c, ord(c)))**
    - Between iterations, we need the script to move from character to character in a word entered by the user. Let's call the word variable theWord and we'll worry about obtaining it later.
    - Now is there a quantity related to the characters in the word that can be updated consistently (with the same operation) for all iterations? The index of the characted in the word fits the bill and the update operation is index = index + 1. Thus we have to add a variable representing the index and use that to extract the characters from the word with c = theWord[index].
    - We also need to set index to 0 for the first iteration.

Our design so far looks like this:

**index = 0**

**while <condition>:**

**c = theWord[index]**

**print("{0}[{1}]".format(c, ord(c)))**

**index += 1**

* + To design the condition, we must analyse the expected sequence of indices. It starts at 0 and ends at len(theWord) - 1. We want the loop to stop after that and can achieve it with the condition index < len(theWord).

We can now write the short program. It also needs to include the reading of the word as user input. It is provided as code snippet [[CS-8]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-des-det-rt). The length of the word is stored in a variable for optimisation (i.e. so that the length is not calculated in every iteration).

###### Example [CS 8] - A loop that prints out the characters of a word

**score**

##### In-class exercise [CE-F]

Write a Python script with a while loop that reads a list of words input by the user and prints them one per line, each word preceded by its ordinal number (1, 2, 3 etc.).

**To DO** > Do this above

[Example solution](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/list_print.py)

##### Example 3 - processing a sequence defined by a sentinel

Let's say we want to collect some sentences from the user and add them to a list, which is then printed at the end.

* + Statement block design:
    - The number of sentences is not predefined so we need to introduce a sentinel (special sentence) that signals the end of the entry process. We could use an empty string but that would be prone to mistakes. Instead let the sentinel be a full stop (.).
    - The action we need to perform on the entered sentences is to add them to a list: sentenceList += [ currSentenceStr ]. So we need a list variable and we need a string to hold the latest sentence input by the user.
    - To create conditions for the next iteration, we need to read in the next sentence: currSentenceStr = input("Enter your next sentence or full stop (.) if finished: ")
    - To create the initial conditions we need to initialise all variables:
      * for the sentence, use the same code as above: currSentenceStr = input("Enter your next sentence or full stop (.) if finished: ")
      * for the list, start with an empty one: sentenceList = []
  + Condition design: this can be a simple check for the sentinel value: currSentenceStr != "."

The full program, including the printing at the end, is given as code snippet [[CS-9]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-des-sent).

###### Example [CS 9] - A loop that collects sentences until a 'stop' value is entered

**sentenceList = []**

**currSentenceStr = input("Enter your next sentence or full stop (.) if finished: ")**

**while currSentenceStr != '.':**

**sentenceList += [ currSentenceStr ]**

**currSentenceStr = input("Enter your next sentence or full stop (.) if finished: ")**

**print("Your sentences:\n------------------\n" + "\n".join(sentenceList))**

##### In-class exercise [CE-G]

Write a Python script that reads in three words separated by whitespace, entered by the user. The program should check the number of words entered and that they contain only alphabetic characters. If the input does not meet these requirements the program should ask the user to try again and repeat the process in a while loop until the input is valid.

Next, the program should print the words, each in a new line and followed by its length in square brackets.

To DO >> do this above

[**Example solution**](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/three_words.py)

#### Condition independent from iteration functionality

All the code snippets in section [*Exiting the loop*](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#ARTCLID000018) ([[CS-2]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-ct), [[CS-3]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-rt), [[CS-4]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-indet-rand) and [[CS-5]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-indet-input)) are examples of loops where the task performed in the while loop statement block is independent of the condition. In all those examples, the task in an individual iteration is to print the number 1.

The statement block core (print(1)) does not define any variables that can be used in the condition. Variables are defined specifically for the purpose of controlling the iteration and updated in the statement block to reflect the iteration requirements.

##### In-class exercise [CE=H]

Write a Python script that fetches a random number between 1 and 3, asks the user to guess it and prints if the guess was correct. It should use a while loop to repeat this 3 times and print, at the end, how many out of three guesses were correct.

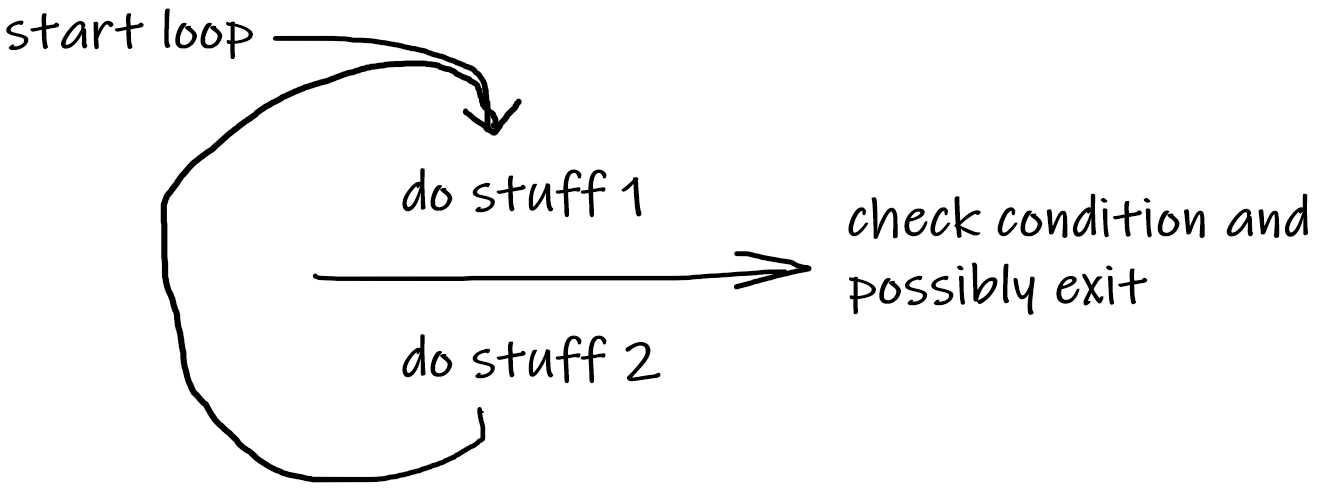
To DO >> do this above

[Example solution](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/independent_cond.py)

### While loop variations

#### Statement block variation

If we step away from the code for a minute we can have a look at iterative processes in a slightly more general form, rather than in terms of Python constructs. Consider the following model of an iterative process:

A general model of iterative processes

Also consider how the iterations of a simple loop such as that in code snippet [[CS-2]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-ct) might be thought of on first analysis (model and pseudocode):

**do stuff 1 set n**

**-----------------> check -----------------> check**

**do stuff 2 print number 1**

The while keyword and condition correspond to the check in the general model i.e. the while loop iterations (condition + statement block) must start where the arrow is. This means that set n for the first iteration of the general model has to happen before the while loop is entered (this is initialisation) and that set n for subsequent iterations must happen at the end of the while statement block (iteration initialisation). After mapping this model onto the Python while loop in [[CS-3]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-det-rt), we get:

**set n (iter 1) n = 0**

**-----------------> check while number < 11:**

**print number 1 print(1)**

**set n (iter 2 etc.) number = number + 1**

We can map any loop onto this model but it is particularly useful for analysing a problem and translating it into a while loop. We will work through several examples, introducing the new constructs where suitable along the way.

#### While loop without initialisation

If *stuff 1* of the model can be packed into the expression that *check* looks at, then there is no need for initialisation before the while loop. In that case the model would look more like:

**-----------------> do stuff 1 and check**

**do stuff 2**

While this can be implemented using functions (which we will be covering later in the module), otherwise self-contained conditionals do not occur often in Python because Python keeps statements and expressions separate (this is not the case in all programming languages), not allowing assignment as part of an expression, which a loop most often needs to keep track of its state using a variable. For loops handle this implicitly and are the more suitable choice of loop in a lot of cases, but let's stay with the while loop for the moment.

Examples of while loops that do not need initialisation can be found though, and the following code snippet represents one.

##### Example [CS10]- While loop with no initialisation

**import random**

**while random.randint(1, 11) != 11:**

**print(1)**

#### In-class exercise [CE-I]

Rewrite the loop in code snippet [[CS-10]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-no-init) to use a value input by the user in the condition, rather than a random value.

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

#### The front-loaded loop statement block (do-while loop)

Let's have a look at a loop that has nothing in the *stuff 2* part of the model.

A user is entering a password and they are allowed 5 mistakes before they get locked out of an account. The stored passwords are hashed, which means that the entered value must be hashed before it is compared with the stored value.

**do stuff 1 ask user to enter password**

**read input password**

**hash the password**

**set attempt count**

**-----------------> check ----------------------------> check**

**do stuff 2 [nothing]**

We know that the check corresponds to the beginning of a while loop, which means that developing the pseudocode for implementation in Python we get:

**set attempt count (1) |**

**ask user to enter password (first time) |--> initialise**

**read input password |**

**hash the password |**

**-----------------> check input vs. stored pwd**

**and that attempt < 6**

**set attempt count (add 1) |**

**ask user to enter password (again) |--> statement**

**read input password | block**

**hash the password |**

this eventually turning into something like this in Python:

##### Example [CS 11]- A while loop with identical initialisation and loop body [CS-11]

**storedHash = hash("abcde")**

**attempt = 1**

**pwd = input("Please enter your password: ")**

**hashed = hash(pwd)**

**while hashed != storedHash and attempt < 6:**

**attempt += 1**

**pwd = input("Please enter your password: ")**

**hashed = hash(pwd)**

With the similarity between the initialisation and statement block (which may contain a lot more code in a real case), it would be cleaner to have the instructions appear only once in the program. A *do-while* loop would be what we need here but Python does not have this construct. The way to emulate it is to give a dummy pass value to the conditional variables to force the first iteration of the loop to execute. After that proper checking of the conditional takes over.

##### Example [CS 12] - Using dummy conditional variable values to emulate a do-while loop

**storedHash = hash("abcde")**

**attempt = 0**

**hashed = hash("") # dummy: hash of "" (not allowed as password)**

**while hashed != storedHash and attempt < 6:**

**attempt += 1**

**pwd = input("Please enter your password: ")**

**hashed = hash(pwd)**

It is often the case that something slightly different, particularly in prompting the user, needs to be done in the first iteration. The dummy variable can be used to check if the iteration is the first one or not. Code snippet [[CS-13]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-dummy-for-diff) is the same as [[CS-12]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-dummy-cond) except that it prints a slightly different prompt the first time around.

##### Example [CS 13] - Using dummy variable value to check if in first iteration

**storedHash = hash("abcde")**

**attempt = 0**

**hashed = hash("") # dummy: hash of "" (not allowed as password)**

**while hashed != storedHash and attempt < 6:**

**attempt += 1**

**if hashed == hash(""):**

**pwd = input("Please enter your password: ")**

**else:**

**pwd = input("Password incorrect. Please try again: ")**

**hashed = hash(pwd)**

A 'do-while' pattern can be recognised from requirements. It emerges from scenarios where the sentinel value needs to be treated the same as any values before it. Password entry is such a case: both the incorrect passwords and the correct password (the sentinel) are not processed further (i.e. are treated exactly the same).

##### In-class exercise[CE-J]

Write a script including a while loop that reads in words from the user, putting them together into a sentence and printing out the sentence when finished. A word with a full stop at the end signals the end but should be included in the sentence.

Write two versions of the script: one that does not employ a dummy conditional variable and one that does.

Sample expected interaction:

--python--

Enter the next word for the sentence (with . at end if last): I  
Enter the next word for the sentence (with . at end if last): am  
Enter the next word for the sentence (with . at end if last): happy.  
Your sentence: "I am happy."

--python--

[EXAMPLE SOLUTION FOR V1](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/do_while_not.py)   [EXAMPLE SOLUTION FOR V2](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/res/files/python/loops/do_while_done.py)

#### The loop statement block with checking in the middle

Let's look at an example where both the *stuff 1* and the *stuff 2* parts of the model are represented by some instructions in the loop statement block. In code snippet [[CS-14]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-mid-check-no-break) the initialisation (corresponding to *stuff 1* of the model) is repeated as the second part of the loop statement block, so that the natural loop can be fitted to a while loop.

##### Example [CS 14] - A while loop implementing a loop with condition naturally occurring mid-block

**numList = []**

**numStr = input("Please enter a number between 1 and 10 or 11 to stop: ") #**

**if not numStr.isnumeric(): #**

**print("The input is not a whole number. It will be ignored.") #--> stuff 1**

**number = 0 #**

**else: #**

**number = int(numStr) #**

**while number != 11: #**

**if number < 1 or number > 10: #**

**print("The number is not in range. It will be ignored.") #--> stuff 2**

**else: #**

**numList += [ number ] #**

**numStr = input("Please enter a number between 1 and 10 or 11 to stop: ") #**

**if not numStr.isnumeric(): #--> stuff 1**

**print("The input is not a whole number. It will be ignored.") #**

**number = 0 #**

**else: #**

**number = int(numStr) #**

**print(numStr)**

In the case of a loop like this, the break construct (see following section) can be used to exit an iteration in the middle, with the actual condition set to True and not really acting as a condition. The code snippet [[CS-15]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-mid-check-with-break) shows how this can be implemented in Python.

##### Example [CS-15] - Using the break construct for mid-block exit from iteration

**numList = []**

**while True:**

**numStr = input("Please enter a number between 1 and 10 or 11 to stop: ") #**

**if not numStr.isnumeric(): #--> stuff 1**

**print("The input is not a whole number. It will be ignored.") #**

**number = 0 #**

**else: #**

**number = int(numStr) #**

**if number == 11: #--> check**

**break # w/ break**

**if number < 1 or number > 10: #**

**print("The number is not in range. It will be ignored.") #--> stuff 2**

**else: #**

**numList += [ number ] #**

**print("Your number list: ", ' '.join(map(str, numList))) #**

##### In-class exercise[CE-K]

Rewrite code snippet [[CS-6]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-while-indet-file) so that obtaining the next number from the file features only once in the code (in the while loop statement block).

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

#### Fluidity of form and transformations

Now that we have seen some statement block forms, it is important to say that in almost all cases it is possible to transform between them, particularly with the use of functions to implement some parts of the functionality. The chosen form will be dictated by optimisation, readability and other considerations.

##### In-class exercise [CE-L]

Rewrite code snippet [[CS-15]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-mid-check-with-break) to subsume 'stuff 2' into the conditional structure in 'stuff 1' and introduce a conditional variable (end) that is prepared in the conditional structure, turning the loop into one with a do-while pattern.

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

#### Additional while loop constructs and their use

The keywords **break, continue** and **else** are used in Python for some additional constructs that can be used with the while loop.

* + break makes the flow of execution 'escape' the while loop and resume from the first instruction outside the loop
  + continue makes the flow of execution 'escape' from the current iteration and resume at condition checking for the subsequent iteration
  + else follows the loop immediately and contains a statement block to be executed after the while loop, but only if it did not come to an end due to a break instruction

##### Using break and else

If a while loop can exit both by the condition being false and by break being called, different behaviour may be required depending on how the exit happened. The else clause at the end of the Python while loop allows the inclusion of a statement block to be executed only if the while loop exits by False condition.

Code snippet [[CS-13]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-dummy-for-diff) is a good example of this. After the looping ends, it would be good to know whether it happened because the user exhausted the attempts of because they entered the correct password. In the example [[CS-16]](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-break-and-else) break is used if the number of password entry attempts has been exhausted. The else clause is executed only when the password is correct and the loop exits as a consequence of that.

###### Example [CS-16] – Using break and else

Advanced so don’t worry if don’t do them… good to know

**storedHash = hash("abcde")**

**attempt = 0**

**hashed = hash("") # dummy: hash of "" (not allowed as password)**

**while hashed != storedHash:**

**attempt += 1**

**if attempt == 7:**

**print("You have run out of tries, sorry.")**

**break**

**if hashed == hash(""):**

**pwd = input("Please enter your password: ")**

**else:**

**pwd = input("Password incorrect. Please try again: ")**

**hashed = hash(pwd)**

**else:**

**print("You are now logged in.")**

rare for while loop to have else statement

###### In-class exercise[CE-M]

Write a while loop with break and else statements that reads words input by the user and prints out the list of collected words at the end. Do not add a word to the list if it contains non-alphabetic characters. Use sentinel values:

* full stop (.) to signal the end of input
* three dashes (---) for cancelling the entire process

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

##### Using continue

Let's say we want to print out all the numbers from 1 to 100, leaving any numbers that contain zeroes. One way to escape from the iteration that we do not want to complete is by calling continue.

###### CS-17 – Example - Printing numbers with no zeroes [CS-17]

**count = 0**

**while count < 101:**

**count += 1**

**if '0' in str(count):**

**continue**

**print(count)**

*(comment - continue means hop out of this particular iteration and don’t execute anything below continue but go on to the next iteration)*

###### In-class exercise[CE-N]

Write a script in Python that uses a while loop with a break and a continue construct. It should read words input by the user, with sentinel value '.' for ending. In the case that the word is made out of alphabetic characters only, it should be converted to uppercase and its characters printed out separated by spaces, otherwise nothing else should be done in that iteration.

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

(Comment - Don’t look at while loop variations except those extra key words unless you want more practice with them – get a feel for different way of setting up an iterating thru stuff)

## For Loops

(Comment – to work with for loops you need to know what a range is)

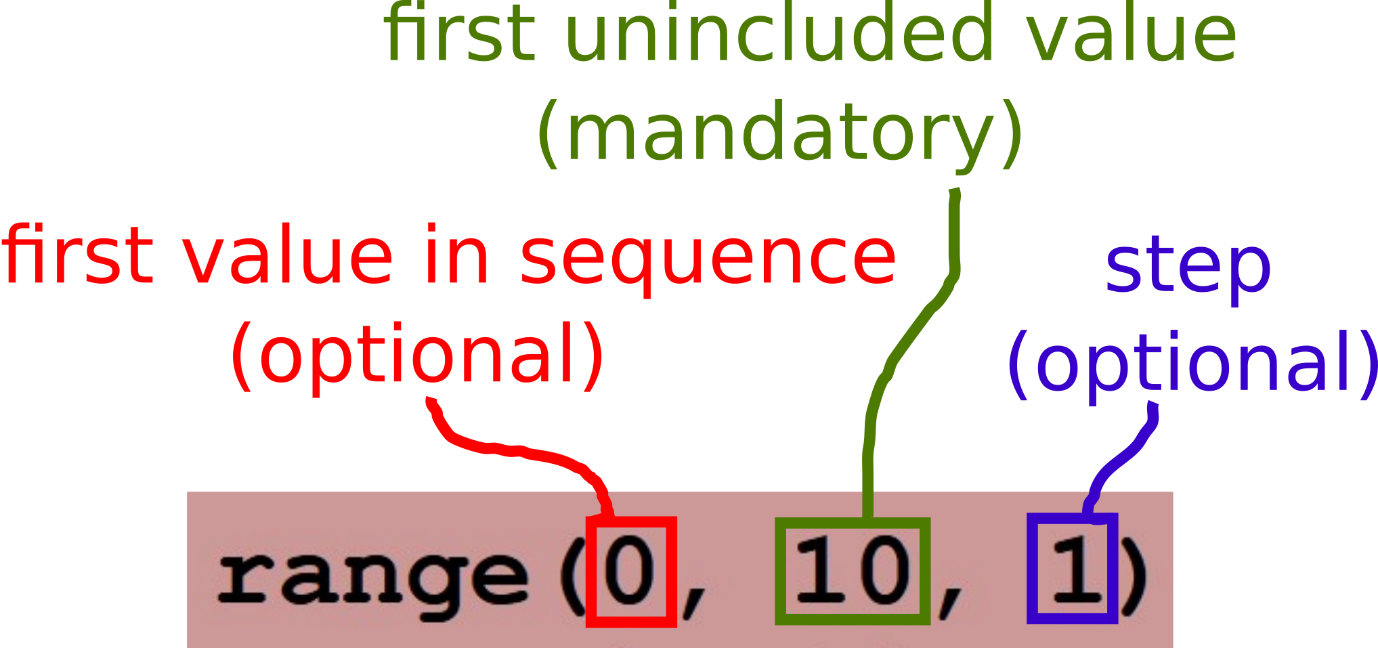
Similar to whiel only difference is we are iterating over pre-set of objects as in numbers ina range or avlues in a list)

### Ranges

The range is an **immutable** sequence type that allows easy definition of **integer sequences**. We introduce it here as for loops in Python are often defined with the use of ranges.

The syntax of a range expression (BNF)

**range(<first value, optional>, <first not-included value>, <step>)**

The syntax of a range expression (example)Examples of ranges [CS-1]

**range(0, 10, 1) # contains 0 1 2 3 4 5 6 7 8 9**

**range(0, 10) # same as above (step 1 is default)**

**range(10) # same as above (start value 0 is default)**

**range(0, 10, 3) # contains 0 3 6 9**

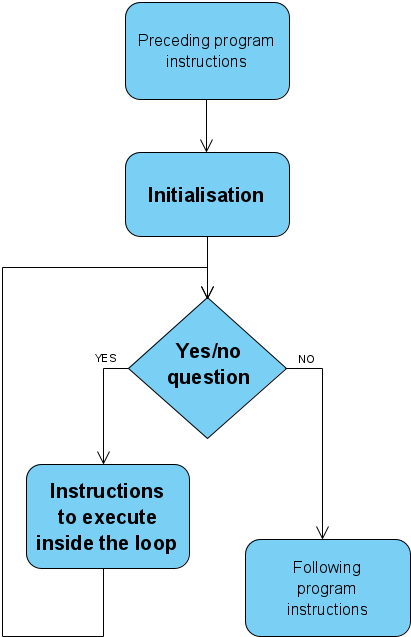
**range(10, 0, -1) # contains 10 9 8 7 6 5 4 3 2 1**

### For Loop syntax in Python

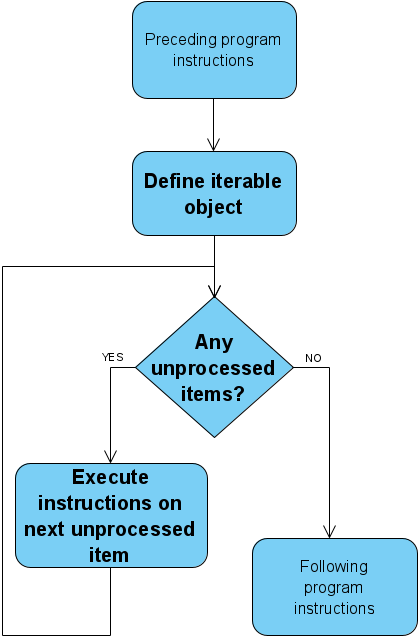
In Python, a for loop iterates over the members of any iterable object. A statement block is defined, like in the while loop, with instructions to be executed at each iteration.

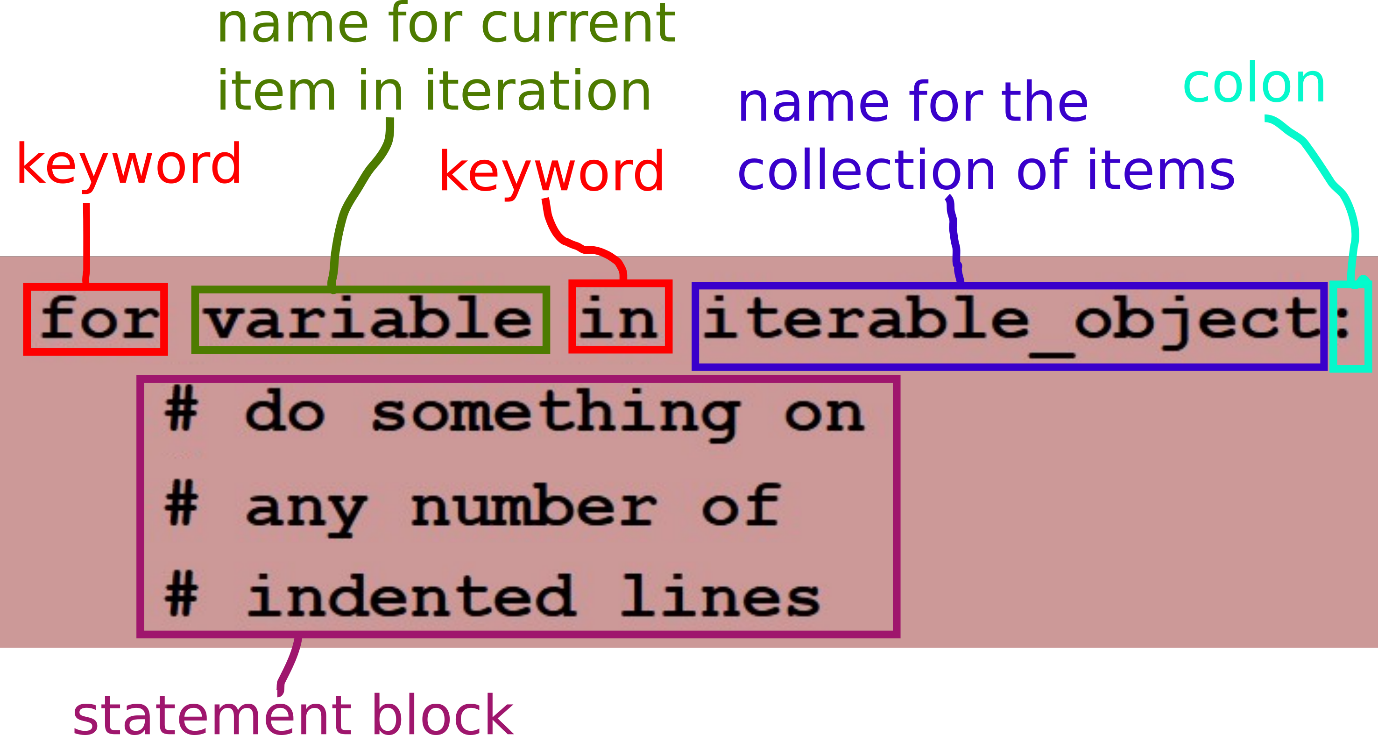
The picture shows the flowchart for a for loop, alongside the generic flow chart.

Generic loop flowchart

Generic loop flowchart

For loop flowchart

For loop flowchart

For loop syntax in Python

### Using for loops

Sequence types are iterable objects in Python. This means that we can design for loops that iterate over any of the sequence types we have covered so far.

In terms of the types mentioned in chapter *While loops*, section [*Exiting the loop*](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#ARTCLID000018), it is the iteration processes for which the iterations are determined before execution starts (either at coding time or at run time) that are suitable for implementation using for loops.

#### Iterating over ranges

###### Example [CS-2] - Iterating over ranges with for loops

**for number in range(0, 10, 1):**

**print(number)**

**for number in range(1, 11, 2):**

**print(number)**

**for number in range(11, 0, -3):**

**print(number)**

##### In-class exercise {CE-A]

Write a script that uses a for loop to print out 11 random integers between 1 and 11.

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

##### In-class exercise {CE-B]

Write a script that uses a for loop to print out all the even integers between 1 and 100, inclusive, separated by spaces.

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

#### Iterating over strings

###### Example [ CS-3] - Iterating over strings with for loops

**for character in "TU Dublin":**

**print(character)**

**word = "computing"**

**for character in word:**

**print("{}: {}".format(word.index(character), character))**

**# try the above with word = "Tallaght"**

**# better solution for above:**

**word = "Tallaght"**

**for index in range(0, len(word)):**

**print("{}: {}".format(index, word[index]))**

##### In-class exercise [CE-C]

Write a Python script that reads in a character string input by the user into a variable called **inputString**. Then, using a for loop, it should invert the order of the characters in the word. Finally, it should print out the string resulting from the inversion, the output of expression inputString[::-1] and the result of comparing the two.

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

#### Iterating over lists

###### Example [CS-4] - Iterating over lists with for loops

**nonMotorised = [ "rollerblades", "bicycle", "scooter", "sail wagon", "legs" ]**

**maxWordLen = max(map(len, nonMotorised))**

**for mode in sorted(nonMotorised):**

**print("--< " + mode.center(maxWordLen, ' ') + " >--")**

##### In-class exercise [CE-D]

Write a Python program that defines a list of items and stores it in a variable. Print out a menu, using the list items as menu items, each on a separate line and preceded by its ordinal number (1, not 0, for the first item etc.). You must use a for loop, iterating over the list rather than over list indices.

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

### Using break, continue and else with for loops

The break, continue and else constructs can be used with a for loop in the same way as they are used with while loops.

#### In-class exercise [ CE-E]

Rewrite code snippet [CS-16](https://jelena-vk-itt.github.io/jvk-tudt-notes/ooswd/lec_8_loops_while.html#py-continue) with a for loop instead of the while loop.

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

Jelena Vasić