Introduction to Python Programming

In this lecture, we'll look at what Python is, why you should start learning Python, introduce Python basics with several exercises to solidify concepts.

This notebook will cover the following topics:

1. What is Python?
2. Why Python?
3. Python Environments
4. Install Python
5. Navigate Notebooks
6. Python Comments
7. Variables and Assignment
8. Data Types
9. Operators
10. Python Libraries and Modules
11. Python Iterables
12. Loops
13. Functions

```
In [ ]:
```

What is Python?

Python is a

- **high-level** *i.e., has strong abstraction from low-level details like the computer's instruction set architecture (ISA) (ISA specifies the behavioiur of machine code; written)*;

- **object oriented** *i.e., programming paradigm based on the concept of "objects" which have state and operations on state*;

- **general purpose**, _i.e., builds software in a wide variety of application domains; versus SQL, which is a domain specific programming language for querying relational databases. You can build web applications, mobile applications, testing, automation, data science, desktop software;

- **interpreted language**,*i.e., interpreter executes statements/source code line by line rather than compiling the whole source file before executing*.

```
In [ ]:
```

```
In [ ]:
```

Why Python?

- General purpose language that is easy to learn due to its clear syntax.
- Quick prototyping.
- Python has a large ecosystem that includes lots of libraries and frameworks (Flask, Django, etc.).
- Python has a huge community. Whenever you get stuck, you can get help from an active community. For example, StackOverflow.
- Python is cross-platform. Python programs can run on Windows, Linux, and macOS.
- Python developers are in high demand.

```
In [ ]:
```

Python Environments

Web-based interactive development environment for notebooks

What is a notebook? A web application for creating and sharing computational documents. It offers a simple, streamlined, document-centric experience.

Notebooks have extension **.ipynb**

- Google Colab
- Jupyter Notebooks
- Anaconda

Integrated Development Environments (IDEs)

What is an IDE? A software application that offers extensive software development abilities, and often consist of a source code editor, build automation tools, and a debugger.

Working with source files with extenion **.py**

- PyCharm
- Visual Studio Code by Microsoft
- Spyder
- Vim
- Emacs
- etc. .

Interactive shell

What is a shell? An interactive shell is any shell process that you use to type commands, and get back output from those commands (Bourne shell, bash or tcsh or zsh)

- IPython

```
In [ ]:
```

Install Python

Install Python in Linux

Before installing Python 3 on your Linux distribution, you check whether Python 3 was already installed by running the following command from the cell (or from the terminal without the magic command !):

```
In [ ]: # !python3 --version
```

Now install a new version of Python:

```
In [ ]: # !sudo apt install python3.10
```

Latest version of Python is ***3.11.4***.

See Upgrade Python to latest version (3.10) on Ubuntu Linux for more help.

```
In [ ]:
```

Navigate Notebooks

```
In [ ]: # This is a cell
        # To run a cell hit Control+Enter
        # To run a cell and move to the next cell hit Shift+Enter
```

```
In [ ]:
```

Change cell type

Here are a few handy shortcuts for changing cell setup:

```
m changes cell to markdown
a inserts cell above
b inserts cell below
dd deletes a cell.
z undoes delete cell
```

```
In [ ]:
```

Code cell

We'll write Python statements in code cells. In addition to writing code, we can also obtain help about Python statements in the cell.

Example. The simplest and most important task you can ask a computer to do is print a message. In Python, we ask a computer to print a message for us by writing `print()` and putting the message inside the parentheses and enclosed in quotation marks.

```
In [ ]: ?print
```

```
In [ ]: print?
```

```
In [ ]: help(print)
```

```
In [ ]: print("Hello world!")
```

```
In [ ]:
```

Python Comments

Comments annotate coding, describing the code.

Use the hash # command to annotate code. The Python interpreter ignores lines annotated with #.

Comments also help other people understand your code.

Also, they help **future you** understand what you coded. Add small explanations for yourself.

But do not go overboard!

```
In [ ]: # This is a comment, Python ignores these lines. Comments in Python begin w

        #Python scripts are executed linearly, line after line, top to bottom

        print("First line to print!")

        # In Python, text (strings), can be defined either with single quotes ' or 
        print('Printing our second line')
```

```
In [ ]:
```

```
In [ ]:
```

Variables and Assignment

Variables are like stickers put on objects.

Every sticker has a unique name written on it, and it can only be on one object at a time.

If desired, more than one sticker can be put on the same object.

Variables form part of the state of a program.

Definition. The state of a program is a mapping of variables to values.

So variables come into existence when they are first assigned values.

The assignment statement Python syntax uses a single equal sign `=` to assign a value to a variable.

Algorithmically the syntax for assigment is `:=`.

In [ ]:
```
# create variables and assign them values
x = 2
y = 17
z = 3
```

In [ ]:
```
```

Variable names

Variables can have complex names like `total_of_people`. In general, never start a variable name with a number and never use spaces in variable names.

See the PEP 8 – Style Guide for Python Code for naming conventions.

In [ ]:
```
# bad
bad name = 5
```

In [ ]:
```
```

Instead, use Python's camel case notation or underscore character in a complex variable name.

In [ ]:
```
# better

badName = 5   # use camel case
bad_name = 5  # or an underscore char
```

In [ ]:
```
```

Variable names should be descriptive to make searching for them easier.

Abbreviations and common words are not searchable.

In [ ]:
```
# bad
CTX_RSP_DICT = {}
sum = 90

# better
CONTEXT_RESPONSE = {}
total_of_people = 90
```

In [ ]:
```
```

In [ ]:
```
```

Keywords

Some words are reserved in Python. They are called keywords.

Don't use them as variables names!

What are these keywords?

In [ ]:
```
# import keyword

# print(keyword.kwlist)
```

In [ ]:
```
 help("keywords")
```

In [ ]:
```

```

Multiple assignment

In Python we can initialise (i.e., create and a assign value) multiple variables in one line.

In [ ]:
```
a,b,c = 10,20,30
print(a,b,c)
```

In [ ]:
```

```

We can also use multiple assignement to interchange values. Suppose we have $m, n \in \mathbb{N}$. If we begin $m := n$ then we have lost the original value of m because it has been overwritten with the value of n. So instead we must first store it and later assign the stored value to n:

```
t := m
m := n
n := t .
```

Updating two variables at the same time at the same time is allowed in Python. To swap the values of m and n using multiple assignment, we write algorithmically:

```
m, n := n, m
```

In [ ]:
```
# try it!
m = 9
n = 98
print(m,n)
m,n = n,m
print(m,n)
```

In [ ]:
```

```

Exercise 1

Write code to swap two numbers using 3 assignment statements.

In [ ]:
```
x = 9
y = 10
print(x,y)

t = x
x = y
y = t

print(x,y)
```

In [ ]:

Data Types

When processing information we need to somehow classify it. Data types are on way to do this. Data types in programming are borrowed from mathematics. We're familiar with the following from mathematics:

|Types|Description| |:---|:---| |$\mathbb{N}$|The natural numbers, or non-negative integers.| |$\mathbb{Z}$|The integers, positive, negative, and zero.| |$\mathbb{Q}$|The rational numbers.| |$\mathbb{R}$|The real numbers.| |$\mathbb{C}$ |The complex numbers|

We might also be familiar with data types in statistics:

|Group|Types|Example values| |:---|:---|:---| |Qualitative| Nominal| Gender, Eye colour, Nationality| | | Ordinal| Ranking in a competition, Economic status, Education level| |Quantitative|Discrete| Age, Marks, Scores, Time| | |Continuous| Height, Weight, Stock option prices|

Python also has its built-in data types, which include numbers, Booleans and strings.

|Data types|Example values| |:---|:---| |int|-2,0,2, 2048| |float|3.14, 2.718, -7.0| |complex|4+7j| |str|"Hello World!", "Don't do it!", '6'| |bool|True, False|

Every type has operations which can be applied to its elements.

In [ ]:

Numbers

There are 3 distinct built-in numeric types. These are integers, floats, and complex numbers. Additional numerics like decimals and fractions can be imported with the following commands:

```
from fractions import Fraction
from decimals import Decimal
```

In [ ]:

`int`

Integers are whole numbers without a decimal point. This includes positive and negative whole numbers as well as zero.

In [ ]:
```
# create (declare & initialise) an int
i = 9
print(type(i))
```

To find out which operations are associated with the data type **`int`**, use the `help` command.

In [ ]:
```
# help(int)
```

In [ ]:
```

```

`float`

Floating-point numbers are written with a decimal point.

In [ ]:
```
# create a float
f = 1.9
print(type(f))
```

Want to know which operations are associated with type **`float`**? Use the `help` command.

In [ ]:
```
# try it!
```

In [ ]:
```

```

`complex`

In [ ]:
```
# complex numbers
c = 5 + 7j
print(type(c))

# print the real and imaginary parts
print(c.real, c.imag)
```

Want to know which operations are associated with type **`complex`**? Use the `help` command.

In [ ]:
```

```

Strings

Besides numbers, Python also has the data type string, which is a finite sequence of characters.

A string is expressed using either single quotes `'...'`, double quotes `"..."` (with the same result) or triple quotes `""" ...""".

Let's look at some examples. We initialise a string with single quotes or double quotes. For example:

In [ ]:
```
message1 = 'This is a string in Python'
message2 = "This is also a string"
```

In [ ]:
```
print(message1)
```

In [ ]:
```
print(message2)
```

If a string contains a single quote, you should place it in double-quotes like this:

In [ ]:
```
message = "It's a string"
print(message)
```

And when a string contains double quotes, you can use the single quotes. For example:

In [ ]:
```
message = '"Beautiful is better than ugly.". Said Tim Peters'
print(message)
```

To escape the quotes, you use the backslash `\`:

In [ ]:
```
message = 'It\'s also a valid string'
print(message)
```

The Python interpreter will treat the backslash character `\` special. If you don't want it to do so, you can use raw strings by adding the letter `r` before the first quote. For example:

In [ ]:
```
message = r'C:\python\bin'
print(message)
```

A string can span multiple lines. Use triple-quotes `"""..."""` or `'''...'''`. For example:

In [ ]:
```
docstr = '''
int: type conversion function
    str: argument to convert to type in
'''

print(docstr)
```

In [ ]:
```
Sometimes, you want to use the values of variables in a string.
```

In [ ]:
```

```

`fstring`

Python `f-strings` allow us to use the values of variables in strings. For example, if we want to use the value of variables `name` and `city` inside a string literal:

In [ ]:
```
name = "Martha"
city = "Windhoek"
```

We place the letter `f` before the opening quotation mark and put the brace around the variables `name` and `city`:

In [ ]:
```
f"Hello, I'm {name}. I was born in {city}."
```

Python will replace `{name}` and `{city}` by the values of the variables.

In [ ]:
```
```

`str.format()`

A slightly older technique one might encounter in older Python code uses the `str.format()` method.

In most cases, the technique is similar to the even older `%`-formatting, with the addition of the {} and with : used instead of %. For example, '%03.2f' can be translated to '{:03.2f}'.

In [ ]:
```
'{2}, {1}, {0}'.format('a', 'b', 'c')
```

In [ ]:
```
'{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
```

In [ ]:
```
c = 3-5j
('The complex number {0} is formed from the real part {0.real} ''and the ima
```

See Common string operations for more string operations.

In [ ]:
```
```

More string methods

String methods include `len(), lstrip(), rstrip(), lower(), upper (), split(), replace('x','y' ), capitalize(), ++ (catenation), * (repeated catenation)` For example the method strip() removes any leading and trailing blanks from the string.

There are many more functions associated with the string object. List them using `help(str)`:

In [ ]:
```
# help(str)
```

Examples:

In [ ]:
```
courseTitle = "  introduction to PYTHON via Design Space_  "
print(len(courseTitle))

print("We can strip away trailing white space to the left")
print(courseTitle.lstrip(' '))

print("We can convert a string to lowercase with lower()")
print(courseTitle.lower())

print("We can convert a string to uppercase with upper()")
print(courseTitle.upper())
print("We can split a string according to whitespaces")
print(courseTitle.split(" "))

print("We can replace substring in a string with a string or character")
print(courseTitle.replace('via', 'by'))


print("We can convert the first letter to upper case with capitalize()")
print(courseTitle.capitalize())

print("We can concatenate two strings")
print("The course is" + courseTitle)

print("We can repeatedly add a string")
print(3 * courseTitle)


print(f"But '{courseTitle}' remains the same")

print("We can sort the characters in a string")
print(sorted(courseTitle))

print("We can reverse a string")
print(courseTitle[::-1])
```

In [ ]:

Exercise 4

Your friend just sent you his part of an essay (see `essay` variable below) but you realise it's a mess! Write some code to correct the contraction (i.e. `doesn't` becomes `does not`), properly capitalise the sentence, and take out all of the times he says totally!

In [ ]:
```
essay = "sometimes the world can be totally messed up but it totally doesn'
```

In [ ]:
```
print(essay
      .replace("doesn't", "does not")
      .capitalize()
      .replace("totally", '')
)
```

Concatenation

We can concatenate two strings literals that are placed next to each other. For example:

In [ ]:
```
greeting = 'Good ' 'Morning!'
print(greeting)
```

We can also concatenate two string variables using the operator `+`. For example:

In [ ]:
```
title = 'More than a glitch '
author = 'Meredith Broussard'

book = title + "by " + author
print(book)
```

Indexing

Since a string is a sequence of characters, you can access its elements by indexing.

The first character in the string has an index of zero.

For example:

In [ ]:
```
print(book[0])
print(book[1])
```

You can also do negative indexing. Python returns the character starting from the end of the string. For example:

In [ ]:
```
print(book[-1])
print(book[-2])
```

Slicing strings

In [ ]:
```
string[start:end]
```

Exercise 3

String comparison

Comparing strings is important. In order to perform tasks like searching or sorting, we need to verify if two strings are equal or not.

For example, when searching a database for a particular value, we need to compare the value we're looking for with the values inside the database.

Python provides string comparison operators. These include:

|Operator|Description| |:---|:---| |<| e.g., $x < y$, is x smaller than y?| |<=| e.g., $x <= y$, is x smaller than or equal to y?| |>| e.g., $x > y$, is x greater than y?| |>=| e.g., $x >= y$, is x greater than or equal to y? | |==| e.g., $x == y$, is x equal to y?| |!=| e.g., $x != y$, is x not equal to y?|

What is the return type when comparing strings?

In [ ]:
```
# answer
a = input("Enter a string: ")
b = input("Enter another string: ")

print(f"{a} < {b}\t\t",   a < b)    # less than?
print(f"{a} <= {b}\t",   a <= b)    # less than or equal?

print(f"{a} > {b}\t\t",   a > b)    # greater than?
print(f"{a} >= {b}\t",   a >= b)    # greater than or equal?

print(f"{a} == {b}\t",  a == b)   # are they equal?
print(f"{a} != {b}\t",  a != b)   # are they not equal?
```

All these expressions return `True` or `False`, which means they're Boolean expressions.

Exercise 2

Question 1. Print the string `another_book = "The Art of Logic by Eugenia Cheng"` backwards.

```
In [ ]: another_book = "The Art of Logic by Eugenia Cheng"
        print(another_book[::-1])
```

```
In [ ]:
```

Question 2. A palindrome is a finite sequence which reads the same forwards and backwards. For example, `[0, 1, 1, 0]` or the conventional example `''Madam I’m Adam''` (ignoring case, punctuation and spaces, to get the string `''madamimadam''`). Write code to detect whether `Madam I'm Adam` is a palindrome.

```
In [ ]: str1 = "Madam I'm Adam"

        # illustrate function chaining
        str1 = str1.casefold().replace("'", "").replace(" ", "")

        # check if string is a palindrome
        str1 == str1[::-1]
```

```
In [ ]: # alternatively
        str1 == "".join((list(reversed(str1))))
```

```
In [ ]:
```

String immutability

Python strings are immutable. This means they cannot be changed once they are created. If you try to update one or more characters in a string, you'll get an error. For example:

```
In [ ]: book[0] = 'T'
```

When we want to modify a string, we need to create a new one from the existing string. For example:

```
In [ ]: print(id(book))
        new_book = 'J' + book[1:]
        print(new_book, id(new_book))
```

```
In [ ]:
```

```
In [ ]:
```

Bool

Boolean algebra consists of two Boolean values are `True` and `False`. And it uses logical operators such as conjunction (AND) denoted as $\wedge$, disjunction (OR) denoted as $\vee$, and the negation (NOT) denoted as $\neg$.

In Python, the Boolean type `bool` is a built-in data type which represents the values `True` and `False`

Type `bool` is a subtype of integers.

Recall the definition of an algebra.

Definition [Algebra] An algebra over a field is a vector space equipped with a bilinear product (Wikipedia).

In [ ]:
```
truth = True
falsehood = False

print(type(truth))
print(type(falsehood))

print(int(truth), ",", int(falsehood))
```

Most variables can be converted to a boolean; usually they are converted to True. For example:

In [ ]:
```
print(bool("potato"))
```

The question is, why does the above return `True`? Because the formula `True` describes all states (because it is true in all states).

As an additional primitive we have `None` which is the null variable

In [ ]:
```
null = None
print(type(null))
```

So what state does `null` describe?

In [ ]:
```
# The null values (0, "", None, and empty lists and dicts) are converted to

print(bool(""))
print(bool(0))
print(bool(null))
print(bool([]))
print(bool({}))
```

In [ ]:
```

```

Logical operators

In [ ]:
```
# help(bool)
```

Python denotes the logical operators for conjunction AND, disjunction OR, and negation NOT as `and, or, not` respectively. These are binary operators which take two operands. The operands in an expression are known as conditions.

| Operator | Symbol | Example | | --- | --- | --- | | AND | `and` | (x > 5 and x < 10) | | OR | `or` | (x > 5 or x < 10) | | NOT | `not` | not(x > 5 and x < 10) |

Generally, these operators are used to represent the truth values of the expressions. For example, `1==1` is `True` whereas `2<1` is `False` (see the above table for more examples).

Let's evaluate a few expressions:

In [ ]:
```
True and True
```

In [ ]:
```
False or False
```

In [ ]:
```
True and False
```

In [ ]:
```
False and True
```

In [ ]:
```

```

In [ ]:
```

```

Exercise 6

A proposition about the elements of some set is a statement about those elements which is ***either true or false*** (though not both). For instance the propositions "3 is odd" and "there are infinitely many even numbers" both happen to be true statements about natural numbers. Of the real numbers, the proposition " π lies within .1 of 3" is false.

Propositions about elements from the same set may be combined by the propositional connectives, which are: negation, $\neg$; conjunction, $\wedge$; disjunction, $\vee$; implication, $\Rightarrow$; equivalence, $\Leftrightarrow$; and exclusive-or, $\not\Leftrightarrow$.

The two Boolean constants $\mathbf{1}$ and $\mathbf{0}$ return the values `True` and `False`, respectively.

Let's recall the truth tables of the propositional connectives. For brevity write the Boolean values `True` and `False` as 1 and 0 which matches the Boolean constants $\mathbf{1}$ and $\mathbf{0}$, respectively. Let p and q be propositions. Write down the truth tables for the following propositional formulae:

a) conjunction: $p \ \wedge \ q$

b) disjunction: $p \ \vee \ q$

c) implication: $p \ \Rightarrow \ q \ \Leftrightarrow \ \neg p \ \vee \ q$

d) equivalence: $p \ \Leftrightarrow \ q \ \equiv \ p = q$

e) exlcusive disjunction (XOR $\oplus$): $p \not\Leftrightarrow q \equiv (p \wedge \neg q) \vee (\neg p \wedge p)$.

In [ ]:
```
# a) conjunction
a = False
for p in [True, False]:
    for q in [True, False]:
        a = (p and q)
        print(int(p), int(q),int(a))
```

Source

In [ ]:

```

```

In [ ]:

```
# b) disjunction
a = False
for p in [True, False]:
    for q in [True, False]:
        a = (p or q)
        print(int(p), int(q),int(a))
```

Source

In [ ]:

```
# c) implication
a = False
for p in [True, False]:
    for q in [True, False]:
        a = ((not p) or q)
        print(int(p), int(q),int(a))
```

Source

In [ ]:
```
# d) equivalence
a = False
for p in [True, False]:
    for q in [True, False]:
        a = (p == q)
        print(int(p), int(q),int(a))
```

In [ ]:
```
# e) exclusive disjunction
a = False
for p in [True, False]:
    for q in [True, False]:
        a = (p and (not q)) or ((not p) and q)
        print(int(p), int(q),int(a))
```

Source

In [ ]:

```
```

Exercise 4

Question 1 Type Conversion

Python is a dynamically-typed language.

This means that the type of the variable does not have to made explicit but can be inferred from the value assigned to a variable.

So in other words, a programmer does not need to declare the type of the variable when they initialise it.

But a programmer can convert the type of a variable into another type using type conversion functions. The following shows the most important ones for now:

- `float(str)` – convert a string to a floating-point number;
- `int(str)` – convert a string to an integer; and
- `bool(val)` – convert a value to a boolean value, either True or False.
- `str(val)` – return the string representation of a value.
- `list(<iterable>)` – return a list representation of an iterable.

Let's see how these functions work.

In [ ]:

```
#
print(float("234.9"))

# convert a string literal to a float
print(float("my name is"))
```

In [ ]:

```
# convert a string made up of numbers to an int
print(int('32'), "\n")

# but this does not work for sequence of characters
print(int('Hello'))
```

In [ ]:

```
print(str(23))
```

In [ ]:

```
```

a) Let's explore converting from one type to another using the functions above.

Use the `input` function to ask the user to enter `int` `a` and a `float` `b`.

Store the result of adding `a` and `b`. Print the result.

What is the type of result?

Can you think of a reason why the output has that type?

In [ ]:
```
a = int(input("Enter an int:"))
print(a)
b = float(input("Enter a float:"))
print(b)

result = a + b

print("result =",result)

print(type(result))


# Answer:
# The `input()` function returns a string, not a number.
# So we use the functions int and float to convert input strings to desired
# Additionally, the result will be a float. Why?
# This has to do with precision and number representation.
# To avoid loss of precision, Python converts the result into a type with b
```

In [ ]:

b) What happens when we use the `int` function with a float value as an argument?

In [ ]:
```
a = 1.7
a_int = int(a)
print(a_int)

# Answer: instead of rounding the float to the nearest integer,
# it truncates (chops off) the decimal portion of the number.
```

In [ ]:

c) Discuss why you think it is important to do type conversion.

In [ ]:
```
"""Answer: In ML, sometimes data is loaded into Pandas as type object.
Therefore we need to convert it to the numeric or string or bool or
whatever type is appropriate in order to work with it. For example,
sometimes the date in a csv file is not in the right format, and we
need to convert it to the type datetime for ease of use.
"""
```

In [ ]:

In [ ]:

Operators

Exercise 5

Question 1 Numeric operators and expressions

We can use the binary operators shown in the table to create numeric expressions. So an expression is a combination of operators and operands which produces some value or result after being interpreted by the Python interpreter.

|Binary operators|Description| |:---|:---| |<, <=, >, >=| comparison| |!=, ==| equality| |+,- |addition and subtraction| |, *@, /, //,%| multiplication, matrix multiplication, division, floor division and remainder*| |*| exponentiation|

Let's do some examples!

In [ ]:

In [ ]:

Python Libraries and Modules

Oftentimes we need to bring in extra methods and functions and even datasets into our development environment.

These could even be methods and functions you've developed yourself.

They are packaged either as libraries or modules.

A module is a file that contains definitions and statements. It often defines classes, functions, and variables intended to be used in other files that import it. So modules can import other modules.

A module can be executed as a script by adding

```
if __name__ == "__main__":
```
at the end of the module.

More complex modules are put together as libraries and have extensive support. For example, the `math` module.

Finally, the Python standard library is a collection of modules that can be used to access built-in functionality.

A library makes everyday tasks more efficient.

Example libraries, SciPy, Pandas, TensorFlow, Matplotlib, Scikit-learn, NumPy, PyTorch, Keras, Flask, Plotly, Seaborn, SymPy.

In [ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import sklearn.datasets
```

In [ ]:
```
# loading datasets
from sklearn.datasets import load_iris
iris_data = load_iris()
# help(iris_data)
print(iris_data.data.shape)
print(iris_data.DESCR)
```

In [ ]:

In [ ]:

In [ ]:

Control Flow

Python has two types of control flow, namely conditional statements and loops.

Conditional statements

Conditional statements direct the flow of execution a program. A conditional statement allows a state update to be performed by cases. What you may be used to writing as a definition by cases

$$\begin{cases} n-2 & \text{if} & n \geq 2 \\ 0 & \text{otherwise} & n = 0 \end{cases}$$

will be written algorithmically as a binary conditional

```
if n>=2:
    n:= n-2
else:
    n:=0
```

Indentation is essential. The Boolean-valued expression $n \geq 2$ is a condition that determines whether a block of code will be executed or not.

`if` statement

```
if condition:
    # execute code if condition is true
```

Note that the if statement syntax uses the colon `:` to indicate the that what follows belongs to the if statement.

In [ ]:
```
# example
city = input("Enter the name of a city:")
if city == "Vatican City":
    print("This is also a country")
```

In [ ]:

`if..else` statement

If you want to perform one action when the condition is true and another when the condition if false, an `if..else` statement will help you do this.

```
if condition:
    # execute code if condition is true
```

```
else:
    # excute code if condition is false
```

In [ ]:
```python
# if statement for case example
n = int(input("Enter an integer: "))

if n >= 2:
    n = n - 2
else:
    n = 0
print(n)
```

In [ ]:

`if..elif..else`

A definition requiring more than two cases is expressed algorithmically using a multi-way conditional

```
if α :
    A
else if β :
    B
else if γ :
    C
else :
    D
```

The fourth branch has the catch-all guard, the negation of the first, second and third branches.

Note that Python uses the keyword `elif` for `else if`.

In [ ]:

In [ ]:
```python
# example
# syntax for multiway conditional
n = int(input("Enter an integer: "))
if n % 2 == 0:
    print("n is divisible 2")
elif n % 3 == 0:
    print("n is divisible by 3")
else:
    print(f"{n} is neither divisible by 2 nor by 3")
```

In [ ]:

In [ ]:

Python Iterables

Lists

Think of a list as a function l that maps indices to data of some type $\mathbb{T}$

$$l : [0, n) \longrightarrow \mathbb{T} \; ,$$

were n is the length of the list.

We'll use the notation $l[0, n)$ to indicate a list.

Python syntax dictates we use the square bracket notation to create a list.

In [ ]:
```
# create an empty list
l = []
```

In [ ]:
```
# create a non-empty list
l = [9,12,8,0, -1, "string", 4+7j]
```

In [ ]:
```
```

List properties

1. Lists are order preserving.

In [ ]:
```
{1,4,3,2} == {1,2,3,4}
```

In [ ]:
```
[1,4,3,2] == [1,2,3,4]
```

In [ ]:
```
```

2. Lists are dynamic (mutable).

We can add and remove elements to a list.

In [ ]:
```
```

3. Lists may store objects of different types.

A list can contain Booleans, strings, complex numbers, and even other lists.

In [ ]:
```
# ask students to create a list with elements of various types
things = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159, 3+1j, [1,2,3]]
```

In [ ]:
```
```

List methods

In [ ]:
```
help(list)
```

List indexing

In [ ]:
```
things[0]
```

```
In [ ]: things[-1]
```

```
In [ ]:
```

Slicing

Another way to access elements in a list is by slicing. To slice a list we specify the name of the list, a start index and a stop index:

`list[start:stop:step]`

```
In [ ]: things[0:2]
```

```
In [ ]:
```

`range(..)`

The `range(start,stop,step)` object produces a sequence of integers from start (inclusive) to stop (exclusive) by step.

For example: `range(i,j)` produces $i, i+1, i+2, \ldots, j-1$

Start defaults to 0, and stop is omitted! Example: `range(4)` produces $0, 1, 2, 3$.

When step is given, it specifies the increment (or decrement).

```
In [ ]: # help(range)
```

Exercise 8

(a) Use the function `range` to print the first 5 positive even numbers.

```
In [ ]: [*range(0,10,2)]
```

```
In [ ]: list(range(0,10,2))
```

```
In [ ]:
```

```
In [ ]:
```

Loops

For loops

A `for` loop allows us to iterate or enumerate elements in a sequence. There is no need to increment a counter in order to go through a sequence.

The syntax for a loop is as follows:

```
for i in <iterable>:
    # do something
```

For example

```
for i in range(0,n):
          print(i)
```

In [ ]:
```
# ???
```

While loops

A `while` loop enumerates items in a sequence while a condition holds. We can use a counter to keep track of iterations.

```
while condition:
    # do something
```

The condition of the while loop is called a guard.

While the guard is true, repeat the block of code an indefinite number of times.

For example, a linear search algorithm:

```
xs = [item_0, item_1,..., item_m]
x := 'key'
ans := False
i := 0
while (i<n):
    ans := (ans or (x=xs[i]))
    i := i + 1
return ans
```

In [ ]:

In [ ]:

In [ ]:

Functions

User-Defined Functions

Functions help us maintain the **DRY** principle, **Don't Repeat Yourself**! Functions avoid duplicating code; instead of rewriting code, we can call a function. In this way we improve code readability, reusability, and manageability.

Creating a function

Use the keyword `def` keyword to define a function as follows:

```
def functionName(argument_1, argument_2, ..., argument_n):

    """
    function docstring: describe the functions inputs,
operation, and outputs
    """

    <function logic, its scope>

    return Type
```

In [ ]:

```

```

Exercise 7

Let's turn the following mathematical functions into Python functions:

(a) $f(x) = x^2$

(b) $g(x) = x^3$

(c) $h(x) = x^4$

In [ ]:

```python
# (a)
def f(x):
    """
    argument(s)/input(s):
        x: a number
    output
        number: square of the argument
    """
    return x**2

print(f(5.0))
print(type(f(5.0)))
```

In [ ]:

```python
# (b)
def g(x:int)->int:
    """function supports type hinting"""
    return x**3

# call the function
g(5.0)

# determine the function return type
type(g(5.0))
```

Note The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc.

In [ ]:

```python
# (c)
def h(x):
    """
    return type of None
    """
    print(x**4)
```

In [ ]:

```
```

In [ ]:

```
```

Lambda expressions

What are lambda expressions? They originate from λ-calculus, a formal system in mathematical logic for expression of computations, developed by Alonso Church (who was Alan Turing's thesis advisor).

λ-calculus is an elegant way of working with application of functions to arguments.

Why use lambda functions? Use once-off and not again.

In [ ]:

```
f = lambda x: x**2
```

In [ ]:

```
f(2)
```

In [ ]:

```
f(20)
```

In [ ]:

```
```

In [ ]:

```
```

References

Python Tutorial

PEP 703 – Making the Global Interpreter Lock Optional in CPython

In [ ]:

```
```