Python_S3_Basics_Preclass

August 5, 2021

1 SCI1022 session 3 pre-class material: The basics of computer programming in Python exemplified with the evaluation of mathematical formulas

In this first workshop we will introduce the **basics** of computer programming using Python. These will be exemplified with computer programs that evaluate mathematical formulas. While a computer program can solve certainly more challenging tasks than a calculator, mathematical formulas are easy to grasp (everyone has had to some extent exposure to them from early education stages) and let one introduce the basic fundamental concepts of computer programming.

In particular, in the course of this workshop, you will learn, among others: * What is a program and what do we mean by programming, * How to write and run a Python program, * How to evaluate complex arithmetic expressions, * How to work with variables, * How to print formatted text and numbers on screen, * How to compute standard mathematical functions in Python, such as e^x or sin x, * How to read input data from the keyboard, * How to find extra information (beyond what we provide to you in the unit notebooks) when you need it.

To this end, we will rely on documents like the present one, a **Jupyter notebook**. Jupyter notebooks will allow us to create and share documents that contain live code (program text), equations, visualizations and narrative text.

2 Our first program: the formula for the vertical motion of a ball

The first Python program that we are going to write in this course will just evaluate a mathematical formula. This formula describes the vertical motion of a ball thrown up from the ground level. The formula provides the vertical position of the ball, referred to as y, as a function of the time t elapsed since the ball was thrown. From Newton's laws of motion, the formula is as follows:

$$y(t) = v_0 t - \frac{1}{2} g t^2,$$

where v_0 denotes the initial velocity at which the ball is thrown, measured in $\frac{m}{s}$ (meters per second), g is (the magnitude of) the acceleration of the ball caused by the force of gravity, measured in $\frac{m}{s^2}$ (meters per second squared), and t denotes elapsed time measured in seconds. We assume that the ball is at the ground level at $t = 0$, $y(0) = 0$.

To determine the time it takes the ball to move upwards and return to the ground again (i.e., $y = 0$), we can look for solutions of the quadratic equation $y(t) = 0$, namely $t = 0$ and $t = 2\frac{v_0}{g}$. In other words, the ball returns to the ground level after $2\frac{v_0}{g}$ seconds. Thus, it actually only makes sense to observe the position of the ball in the time interval $[0, 2\frac{v_0}{g}]$.

2.1 Using a Python program as a calculator

Our first Python program will evaluate the formula above for specific values of v_0, g, and t. In other words, we will use Python as a calculator. However, we note that Python is waaaay more expressive and can certainly solve significantly more challenging problems than a mere calculator. For example, choosing $v_0 = 10\frac{m}{s}$, and $g = 9.81\frac{m}{s^2}$, we have that the ball will come back to the ground level after approximately $2\frac{v_0}{g} \approx 2$ seconds. This basically implies that our interest is in the $[0, 2]$ time interval.

Let us assume that we want to compute the position of the ball after $t = 1.2$ seconds. If we replace the value of t in the formula above, we get the arithmetic expression

$$y(1.2) = 10 \cdot 1.2 - \frac{1}{2} \cdot 9.81 \cdot 1.2^2$$

This arithmetic expression can be first **evaluated**, and then **printed on screen**, using the following single-line Python program:

```
print(10*1.2-0.5*9.81*1.2**2)
```

In Python (as in most programming languages), the standard summation, subtraction, multiplication and division operators are written as `+`, `-`, `*` and `/`, respectively. In order to raise a number (`1.2` in our example) to the power of another number (`2` in our example), we have to use the exponentiation operator, denoted with two asterisks `**` in Python. We will later discuss different ways of running our single-line program, but let us first define what is a **program**, and what do we mean by **programming**.

2.2 What is a program?

A program is a sequence of instructions in a programming language that specifies how to perform a computation. The computation might be the solution to a mathematical problem, such as solving a linear system of equations via Gaussian elimination, or a symbolic computation, such as searching and replacing text in a document, processing an image or a video, etc. The program text, which contains the sequence of instructions, is stored in one or more text files.

While the instructions in a programming language may look somewhat similar to English, they are way more simpler. The number of reserved words (e.g., `print` in our example above) and associated instructions is very limited, so that in order to perform a complicated computation, we have to use a large number of these instructions. The details look different in different languages, but **a few instructions types** appear in just about every programming language:

1. **Input**: Get data from the keyboard, a file, the network, or some other device.
2. **Output**: Display data on the screen, save it in a file, or send it over the network.
3. **Math**: Perform basic mathematical operations like addition and multiplication.

4. **Conditional execution** (e.g., `if` statement): Check for certain conditions and change the behavior of the program depending on the condition.
5. **Repetition**: (e.g., `while` and `for` loops): Perform some action repeatedly, usually with some variation among repetitions.

Along the course, we will go over these types of instructions step by step. You might be able to recognize some of these types of instructions in our single-line program above. **Every program you will ever develop/use, no matter how complicated, is made up of instructions that look pretty much like these.**

2.3 What do we mean by programming?

Programming is, naturally, the process of writing programs. This process is, however, much more challenging than merely writing the correct instructions in a file. Programming is a **problem-solving** skill in itself. It involves the ability to formulate problems, think about solutions, and express a solution clearly and accurately. In a nutshell, it involves four basic steps:

1. To understand how a problem can be formulated computationally. We give a sequence of instructions to the computer in order to solve it. This is clearly the most difficult step. You can think of this step as the process of breaking a large, complex task, into smaller and smaller subtasks until the subtasks are simple enough to be expressed in terms of the five type of instructions mentioned above.
2. To express this sequence of instructions correctly in a computer language and store the corresponding text in a file (the program). This is normally the easiest part.
3. To check the validity of the results obtained by the program. We must test the program.
4. To systematically track down errors and fix them (this is referred to as **debugging**). Usually, the results are not as expected, and this is why we need this fourth phase.

Mastering these four steps requires a significant amount of training, which means writing many programs and getting the programs to produce the expected results.

2.4 Writing Python programs

There are several alternative types of tools that can be used for writing Python programs:

1. A Jupyter notebook (this document).
2. A plain text editor.
3. An Integrated Development Environment (IDE) with a text editor.
4. Cloud services accessed via a web browser, such as, e.g., Python anywhere or Cocalc, or Google Colaboratory.

What you will write will be the same in either of these types of tools, the difference lies in how you run the program. In this course, we will mainly use Jupyter notebooks on your local computer, like this document, to write, run, and deliver Python programs. It is out of the scope of the course to discuss all possible tools for writing Python programs. For the interested reader, recommended tools of type 2 include Notepad++ for Windows, Atom and TextWrangler for MacOS X, and Gedit, Atom, and Sublime Text (for Linux). The dominating tool of type 3 for Python programs is called Spyder, and it is available on either Windows, Mac, or Linux. Some of the more advanced

plain text editors (e.g., Atom, or Sublime Text) can be converted into IDEs using the appropriate configuration.

2.5 Running Python programs

In Jupyter notebooks, we write program text into code cells, and then left-click the "Run" button at the tool bar located at the top of the page to run the code within the cell. Right below this text you can see a code cell which contains our one-line program above. Code cells, in contrast to text cells, are labeled with `In [ ]:` right at the left of the first line of code within the cell. **Run it!** (Note that you first have to left-click either on the interior or the left-hand-side margin of the cell, to put focus on it, before clicking on the "Run" button).

```
[ ]: print(10*1.2-0.5*9.81*1.2**2)
```

If everything went well, you should see the output of the program, `4.9368` in this case, right below the cell. Alternatively, in order to execute a code cell, you can left-click on it and then hit `Shift-Enter`, which stands for: *"While holding pressed the* ***`Shift`*** *keyboard key, press the keyboard key labelled as* ***`Enter`****"*.

Alternatively, one may type the single line in our program in a plain text editor/IDE, and save the line to a file, named, say `ball.py` (`.py` is the standard extension for Python program text files). The action required to run this file depends on the tool that you use in order to run programs. For example:

- In a Terminal window (e.g., the `git bash` terminal window in Windows, or the Terminal app in MacOS X), you move the current directory to the directory where `ball.py` is located. Then type the `python ball.py` command, where `python` is the Python interpreter the program that reads and executes Python code.
- In Spyder, you choose *Run* from the *Run* pull-down menu.

The output is, again, `4.9368`, and appears:

- Right after the `python ball.py` command in a terminal window.
- In the lower right window in Spyder.

Suppose that now we want to evaluate the formula for $v_0 = 1\frac{m}{s}$ and $t = 0.1$ seconds. In the (void) code cell below, write the program text that carries out such an operation, and prints the result on screen. Then run the cell!

```
[ ]:
```

2.6 Natural versus programming languages

Natural languages are those that people use to communicate, like English. They have evolved across many years. On the other hand, formal languages are designed by people for specific applications in mind. **Programming languages are formal languages that have been designed to express computations.**

Programming languages tend to have **very strict syntax rules** that govern the structure of the statements. Syntax rules come in two flavors: tokens and structure. Tokens are the words, numbers

and symbols that make up the statement in the programming language. For example, the expression `2+3` has three tokens. On the other hand, structure refers to the way the tokens can be combined. For example, `3 * / 4` is illegal, because even if all tokens are legal, you cannot have `/` right after `*`.

Programming languages are designed to be (almost) **ambiguity-free**, which means that any statement has only one meaning, regardless of context. Besides, programming languages are **more concise and less redundant**. They do not have to cope with ambiguity.

To put an example, if one types our one-line formula evaluation program as:

```
[ ]: write(10*1.2-0.5*9.81*1.2^2)
```

it is reasonable to think that most humans would interpret `write` as `print` as the same thing, and many would also interpret `1.2^2` as 1.2^2. In Python, however, `write` is not defined and thus it is an error to use it. On the other hand, `1.2^2` means something very different from the exponentiation `1.2**2`. At this point, you may want to run the code cell right above to see what happens.

2.7 Warning about typing program text. Errors and debugging

Our communication with the computer using a programming language must be precise, without a single error (e.g., in spelling or punctuation). The programmer must type the program very carefully, paying attention to the correctness of every character. For example, you will see along the course that blank spaces are crucial in Python (e.g., in order to indent a new code block inside an existing code block). It is a good practice to always count them and type them in correctly.

Any error, no matter how small, can affect to the result of the program. There are, essentially, three kind of errors in a program:

- **Syntax errors**. This kind of errors are produced whenever one violates the rules underlying the structure of a program. For example, parenthesis come in pairs. If you open a parenthesis, and you forget to close it, then you have made a syntax error. Another example has been already shown above, i.e., to use `write` instead of `print`. **If there is a syntax error in your program, Python will generate an error message, and your program will stop, you will not able to run it.** During your first weeks with programming you will likely make a lot of syntax errors and spend some time tracking down them. However, you will experience that the more you practice, the less syntax errors you will made, and the faster you will find them.
- **Run-time errors**. This kind of errors are produced after the program has started running. These errors are also called exceptions, because they usually indicate that something exceptional has happened. Dividing a number by zero is an example of a run-time error. These errors are much less likely than syntax errors, especially while one is walking the first steps with programming.
- **Semantic errors**. If your program has a semantic error, then it will completely run without any error message, but it won't generate the expected result. The program is doing something different to what it was intended. The issue in this case is that what we told it to do is not what we actually want it to do. Semantic errors are the hardest errors to find and resolve,

as they require one to work backward by looking at the output of the program and trying to figure out what it is doing.

An error in a program is known as a **bug**, and the process of locating and removing bugs is called **debugging**. Many look at debugging as the most difficult and challenging part of computer programming. Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. The origin of the strange terms bug and debugging can be found in Wikipedia.

2.8 Verifying the result

It might sound obvious, but we **always** have to find a way to verify that the output of the program that we have written is correct. As you walk your first steps in programming, you will experience that the output of your program is frequently wrong, and you have to search for errors. For our single-line program, we can use a calculator to evaluate the mathematical formula for $v_0 = 10$ and $t = 1.2$, and verify that the result obtained with the calculator matches the one produced with the program.

2.9 Using variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a programmer-defined name that refers to an object (for instance, a number). Let us use variables in our single-line program.

Let us assume that we want to evaluate our formula for different values of t. For each new value of t, we have to perform changes at two different places in our program. Changing the value of v_0 is simpler, as it only appears once in the expression. However, note that it is not that difficult to modify the wrong number, especially if we do not remember which was the last value that we used for v_0.

Such kind of modifications will be easier if we express the formula in terms of variables (symbols) instead of numerical values. The program text for our alternative, variable-based, may be written as:

[ ]:
```
v0 = 10
g = 9.81
t = 1.2
y = v0*t - 0.5*g*t**2
print(y)
```

Each of the four first lines in the code cell above is called an **assignment statement**. An assignment statement creates a new variable and gives it a value. The value might be a typed number, as e.g., `10` in the first line, or the result of the evaluation of an arithmetic expression involving already defined variables (as `v0*t - 0.5*g*t**2`).

2.10 Assignment versus equality. Statements versus expressions

As seen in the previous section, variable assignment in Python is denoted with the `=` operator. The `x=3` assignment means *"Let us define x to have the value 1."* or *"Let us denote with the symbol x the number 1."* Thus, **the assignment operator = in a computer program does not have the same meaning as the equality sign = in mathematics**. For example, consider the following assignment statement:

```
y=y+5
```

This statement is mathematically false (a number can never be equal to itself plus 5). However, **in a program it just means that we evaluate the right-hand side expression and assign it to the variable `y`**. That is, we first get the value of `y` (whatever it is) and add `5` to it. The value of this expression is assigned to `y`, so that the previous value of `y` is lost. As an exercise, run the code cell below and be sure that you understand the result of each statement as printed on screen.

[ ]:
```
a=5
print(a)
a=a-3
print(a)
a=a*a
print(a)
```

Assignment statements must not be confused with equality, which in Python, is denoted with the `==` operator. When we write an expression like `3==4` or `v0==10`, we are actually asking: *is `3` equal to `4`?* or *is `10` the value of the variable `v0`?* The answer to this question is the result of evaluating the expression. You can check that executing the two code cells below.

[ ]:
```
3 == 4
```

[ ]:
```
v0 == 10
```

At this point, it is important that you note and understand the difference among expressions and statements:

- An **expression** is a combination of numbers, variables, and operators, **which has a value** (e.g., `n+25` or `3==4`). The first returns a number (e.g., `35`) and the second `False`. When you type an expression in a code cell, and run it, the (python interpreter under the) Jupyter notebook evaluates it, which means that it finds the value of the expression, and shows the result right below of the cell with an `"Out[...]"` label right at the left. (Check that for the two expressions above!).
- An **statement** is a unit of code **which has an effect**, like creating a variable or displaying a value but not a value. Examples of these two statements are given in the two code cells below.

[ ]:
```
x=1
```

[ ]:
```
print(x)
```

When you type an statement in a code cell, and run it, the Jupyter notebook executes the statement, but, in general, **statements do not have values**. Indeed, this might be observed, e.g., for the two cells above: there is no `"Out[...]"` label right at the left of the output (if any) after running the cell with the statement.

2.11 Variable names

Variable names have an important impact on the readability and correctness of programs. Experienced programmers choose **names that are meaningful**, i.e., names that **reveal intention**. By solely reading the name of the variable it should be easy to grasp the meaning of the variable, like `initial_velocity` or `position`.

Variable names can be as long as you like. They can contain both letters and numbers, but they cannot begin with a number. Python distinguishes between upper and lower case, so the variable name `G` is always different from `g`. **In any case, it is conventional to use only lower case for variable names.** The underscore character `_` can appear in a variable name. It is frequently used in names with multiple words, such as `initial_velocity`, or `beam_length`.

Our advice for choosing variable names involves two basic, easy-to-remember rules:

1. Whenever the variable represents a mathematical symbol, we use the symbol as variable name. In our example program, y, g, and v_0 in our mathematical formula become `y`, `g`, and `v0` in the program. **Variable names similar to their mathematical symbols increases the readability of the program and eases error detection.**
2. If the variable does not have a match in the mathematical description of the problem, use a descriptive name. For example, if our variable represents traction force, we could call it `traction` or `traction_force`.

2.12 Keywords

Certain words are reserved in Python, because they are needed to build up the language. These reserved words are referred to as **keywords**, and **cannot be used as variable names**. Some examples of keywords include `False`, `break`, `return`, `for` or `while`. Click here for a full list. Fortunately, you don't have to memorize all the keywords in the list. In most development environments, and Jupyter notebooks are not the exception, keywords are displayed in a different color within code cells. For example, in the code cell below, the keyword `False` is displayed differently than the user-defined variable name `x`.

```
[ ]: x = False
```

2.13 Comments

As programs get larger, and more complicated, they get more difficult to read. Programming languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing and why (especially if you did not write it yourself, or you wrote it a long time ago).

For this reason, it is wise to have **informative** notes written in natural language along with the program text. These notes are called **comments**. Comments in Python start with the # character. Everything after this character up to the end of the line is ignored when the program is run.

Comments are most useful when they document aspects of the program that are not obvious to infer solely from the program text, such as what mathematical variables mean, what is their purpose, a quick overview of the problem-solving strategy as reflected in the next block of statements, or general ideas about what the program is doing.

An example of the program above with explanatory comments is as follows:

[ ]:
```
# Program which computes vertical position of a ball thrown from the ground␣
↪level as a function of elapsed time.
v0 = 10   # Initial velocity (m/s)
g = 9.81  # Module of the acceleration of gravity (m/s**2)
t = 1.2   # Elapsed time (s)
y = v0*t - 0.5*g*t**2 # Vertical position/Distance to the ground level (m)
print(y)
```

Note that a comment may either appear on a line by itself (as in the first line of the code cell above), or at the end of a line (lines 2, 3, 4, and 5).

Never repeat with words what the program statements already clearly express. For example, this comment is redundant and useless:

[ ]:
```
v0 = 10 # Assign the value 10 to the variable v0
```

Finally, it worths mentioning that there is a trade-off among variable names and comments. Good variable names can reduce the need for comments, but, on the other hand, large variable names can make complex, hard to read expressions.

3 Another example of formula evaluation: Celsius-Fahrenheit conversion

The next example involves the formula for converting temperature measured in Celsius degrees to the corresponding value in Fahrenheit degrees, namely

$$F = \frac{9}{5}C + 32$$

where the symbol C denotes the amount of Celsius degrees, and F the converted value in Fahrenheit degrees.

Let us write a computer program that, given C, computes F using (i.e., evaluating) this formula. A possible program is as follows:

[ ]:
```
C=21          # Temperature in Celsius degrees
F=(9/5)*C+32 # Compute temperature in Fahrenheit degrees, given its value in␣
↪Celsius degrees
```

```
print(F)
```

In the expression above, the parentheses around `9/5` are not strictly needed. In other words, `(9/5)*C+32` leads to the same result as `9/5*C+32`. (*Check it yourself!*). However, parentheses remove any doubt that `9/5*C` could mean `9/(5*C)`. (The section entitled "Order of operators" below contains a detailed explanation of arithmetic operators and precedence rules in Python.)

To verify our program, we can just evaluate the formula on a calculator, $\frac{9}{5} \cdot 21 + 32$, which returns the value 69.8, as our program in the cell above.

3.1 Objects and types in Python. `int`, `float` and `str`

When we write an assignment statement like:

```
C = 21
```

Python recognizes the number `21` as an integer number and creates an `int` object (for integer) that is able to hold an integer value inside, 21 in our example. The variable `C` acts as a name for that object. Similarly, if we write `C=23.43`, Python identifies `23.43` as a real number, creates a `float` object (for floating-point) holding the value 23.43 inside, and lets `C` be the name for this object.

At this point, you do not need to know what an object really is nor how it is internally organized. It suffices to know that each object belongs to a collection, **the so-called type of the object**. You can think of a type as a category of interrelated objects. In our example, `int` is the type of the object which holds the value `21`, and `float` is the type of the object which holds the real number `23.43`.

If one is unsure about the type of an object, one can use the Python built-in function `type`, as in the code cells below. We will get to functions later on in this course, in the meantime you can think of a function as a black-box to which you pass an object and you get another object in return. Run them!

[ ]:
```
print(type(21))
```

[ ]:
```
print(type(21.0))
```

[ ]:
```
print(type(23.43))
```

In these results, the word "class" is used in the sense of a category. Recall from above that a type is a category of objects. Interestingly enough, note that the numbers `21` and `21.0` lead to objects of different types in Python, although they are identical numbers in mathematics.

Apart from the types representing integer (`int`) and real (`float`) numbers, there are many other object types in Python. For example, we can use either double or single quotation marks and put characters (numbers, letters, symbols, etc.) inside, like `"This is a text"`, `'we have seen so far 2 different object types'`, or `"32"`. Python recognizes these as objects of type `str`. This type name comes from the term "string", as these objects represent a set of characters struck together.

Using objects of type `str`, and `print`, we can print human-readable messages on screen, and not only numbers, as we did so far. This is illustrated in the code cells below, which you can execute.

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

[ ]:
```
print('We are learning the basics of programming in SCI1022')
```

Note that the quotation marks do not get printed on screen. They are actually used to delimit the beginning and end of the text, and are not part of it.

3.2 Type conversion

In most cases, one can work with variables in Python without bothering at all about the types of the objects associated to these variables. In some circumstances, though, one may need to be aware of the types of the objects, most typically when you operate with objects of different types.

For example, the `+` operator, when used with objects of type `str`, performs **string concatenation**, which means that it generates a new string by putting one string after the other, as illustrated in the following cell:

[ ]:
```
print("Hello," + " world!")
```

However, if we try to concatenate, say, a string and a integer number, as in the following code cell

[ ]:
```
print("Hello," + 365)
```

then we get a **runtime** error with a clear message: objects of type `str` cannot be concatenated to objects of type `int`. If we want this operation to be legal we have to convert `365` into an object of type `str`. This is done in Python as follows:

[ ]:
```
print("Hello," + str(365))
```

In general, one can convert an object `o` to type `MyType` by writing `MyType(o)`, **if it makes sense to do the conversion.** (In the example above `MyType` is `str` and `o` is `365`.)

Other examples of (legal) type conversions are as follows:

[ ]:
```
float("3.56")
```

[ ]:
```
str(3.1416)
```

[ ]:
```
int("354")
```

[ ]:
```
float(3)
```

[ ]:
```
int(4.65)
```

Note that the last conversion from `float` to `int` strips off the decimals. A more accurate conversion can be achieved with the help of the `round` function, which rounds the real number to the closest integer number

[ ]:
```
round(4.56)
```

For obvious reasons, the following type conversion is not legal:

```
[ ]: float("Hello!")
```

3.3 Order of operators

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention for precedence rules. The acronym **PEMDAS** is useful to remember these rules:

- (**P**)arentheses have the highest precedence and thus can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is `4`, and `(1+1)**(5-2)` is `8`.
- (**E**)xponentiation has the next highest precedence, so `1 + 2**3` is `9`, not `27`, and `2 * 3**2` is `18`, not `36`.
- (**M**)ultiplication and (**D**)ivision have higher precedence than (**A**)ddition and (**S**)ubtraction. So `2*3-1` is `5`, not `4`, and `6+4/2` is `8`, not `5`.
- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression `degrees / 2 * pi`, the division happens first and the result is multiplied by `pi` . To divide by 2π, you can use `degrees/(2*pi)` or write `degrees / 2 / pi`.

As a general recommendation, it is in most cases better to use parenthesis to make an expression easier to read, even if the parentheses do not change the result. Besides, you do not have to remember the precedence rules!

3.4 Formatting text and numbers (f-strings)

When one writes a program and displays a result on screen, it is convenient to generate human-readable, informative messages with all the necessary context incorporated to the message. For example, instead of just printing the numerical value of `y` in our introductory program, it is more convenient to write a more informative text, typically something like

```
At t=1.2 seconds, the height of the ball is 4.94 meters.
```

There are three ways in Python for formatting text and numbers: * Printf syntax (also known as %-formatting) * Format string syntax * f-strings (also called formatted string literals)

In this course, we will restrict ourselves to the usage of f-strings, as they are indeed the faster and most modern approach.

In a nutshell, an f-string is an special kind of string that has an `f` at the beginning and curly braces containing expressions that are to be replaced with their values. Our complete program, where text and numbers are mixed in the output is as follows:

```
[ ]: # Program which computes vertical position of a ball thrown from the ground␣
     ↪level as a function of elapsed time.
     v0 = 10   # Initial velocity (m/s)
     g = 9.81  # Module of the acceleration of gravity (m/s**2)
     t = 1.2   # Elapsed time (s)
```

```
y = v0*t - 0.5*g*t**2 # Vertical position/Distance to the ground level (m)
print(f"At t={t} seconds, the height of the ball is {y} meters.")
```

Note that the string being printed starts with an `f`, and that we have put two expressions (the variables `t` and `y`) within curly brackets, which have been replaced by their values. We can put any expression within the curly brackets, such as an arithmetic expression:

[ ]:
```
print(f"2*2 is equal to {2*2}")
```

A somewhat more advanced feature of f-strings is that they accept the so-called format specifications to be attached to the expressions within curly braces for additional control of output strings. These format modifiers take the form `{expression:format_specifier}`. In our example, format specifiers can be used, e.g., in order to control the number of fractional digits to be printed on screen:

[ ]:
```
print(f"At t={t} seconds, the height of the ball is {y:.2f} meters.")
```

The `.2f` format specifier means that we want to display a floating point number in decimal notation with only 2 decimals. In this course, we will restrict to the usage of the most simple format specifiers. The interested reader can use this section of the Python docs to learn more about Python's available format specs.

4 Standard mathematical functions. The `math` built-in module

The evaluation of formulas frequently involves standard mathematical functions such as `sin`, `cos`, `tan`, `exp`, `log`, etc. Programming languages provide ready-to-use functionality for evaluating such functions. In principle, one could write a program for evaluating, say, the logarithm of a number, but how to do this in an accurate and efficient way is not obvious. Experts in numerical computing have worked for decades on this problem and implemented the best recipes in pieces of software that it is better to reuse.

Python has the so-called `math` module that provides most of the familiar mathematical functions. You can think of a **module** as a file with program text that contains a set of related functions and constants.

4.1 Example: the square root function

4.1.1 Problem

Let us assume that we want to use our formula describing vertical motion of a ball thrown from the ground level in order to determine at which time the ball reaches a given height y_c. The answer is straightforward, it just involves solving the following quadratic equation for the unknown t:

$$y_c = v_0 t - \frac{1}{2} g t^2.$$

Using the well-known formula for the two solutions of a quadratic function, we find

$$t_1 = \frac{v_0 - \sqrt{v_0^2 - 2gy_c}}{g} \quad \text{and} \quad t_2 = \frac{v_0 + \sqrt{v_0^2 - 2gy_c}}{g}$$

The ball reaches y_c on its way up at t_1 and on its way down at t_2. Note that, in order to compute both t_1 and t_2 we need to be able to compute the square root of a number in our program.

4.1.2 The import statement. How to use a function defined in the `math` module

Before we want to use the functions in a module, the `math` module in our particular case, we have to import it with an import statement

```
import math
```

This statement creates a **module object** which contains all functions (and variables) defined in the module. If you want to access one of the functions of the module, you have to specify the name of the module and the name of the function, separated by a dot in between. This format is called **dot notation**, or **fully-qualified** names. For example, the name of the square root function in the `math` module is `sqrt`. Thus, we have to use `math.sqrt` if we want to access that function in our program.

4.1.3 The program

The use of the `sqrt` function of the math module is illustrated in the following program, that solves the problem stated above:

[ ]:
```
# Program which computes the time at which a ball thrown from the ground level␣
↪reaches a given height yc
import math
v0 = 10      # Initial velocity (m/s)
g  = 9.81    # Module of the acceleration of gravity (m/s**2)
yc = 3.5     # Given height (m)
t1  = (v0-math.sqrt(v0**2-2*g*yc))/g # Way up time
t2  = (v0+math.sqrt(v0**2-2*g*yc))/g # Way down time
print(f"The ball reaches a height of y={yc:.2f} meters at t1={t1:.2f} and␣
↪t2={t2:.2f} seconds.")
```

4.1.4 Different ways of importing functions from a module

So far we have seen that, in order to import a module, say `math`, we first write

```
import math
```

and then access a function defined in the module using dot notation as, e.g., in:

```
x = math.sqrt(y)
```

However, Python offers alternative ways in order to import functions from a module. One can skip dot notation using an statement of the form `from module import function`. This statement lets us access a function named `function` from a module named `module` **directly using the name of the function**. This is illustrated in the following code cell:

[ ]:
```
from math import sqrt
print(sqrt(4))
```

Note that we do not longer have to use ***`math.sqrt`*** *in order to access the* ***`sqrt`*** *function.* This syntax allows us to import more than one function at once:

[ ]:
```
from math import sqrt, exp, log10
print(sqrt(4))
print(exp(2))
print(log10(1000))
```

Even more than that, we can import all functions and constants of the `math` module using the asterisk (`*`) syntax

`from math import *`

All the functions and constants in the `math` module can be found listed here. This includes the famous numbers `e` and `pi`, as illustrated in the code cell below:

[ ]:
```
from math import *
print(pi)
print(cos(pi))
print(exp(2))
print(e**2)
```

While importing all functions from a module is more pleasant than importing them one by one, it nevertheless results in a myriad of extra names in our program that most probably we will not use. **As a general recommendation, it is best to only import those functions that are strictly used in the program.**

Finally, it is worth mentioning that imported modules and functions can be given new names, as illustrated in the following code cell:

[ ]:
```
# Creates a new module object named m that contains all functions and constants␣
↪of the math module
import math as m
print(m.cos(m.pi))

# We give the log10 function a new name, i.e., ln10
from math import log10 as ln10
print(ln10(1000))

# We give the sin, cos functions a new name at once
from math import sin as s, cos as c
print(s(1.0)**2+c(1.0)**2)
```

4.2 Another example: three alternative ways of computing the `sinhx` function

The mathematical function sinh(x) is defined as

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x})$$

In Python, we have three alternative ways to compute this function:

1. Use the `sinh` function of the `math` module.
2. Evaluating the right-hand-side of the formula above with the aid of the `exp` function of the `math` module.
3. Using the constant `e` (imported from the `math` module) and the exponentiation operator `**`.

This three approaches are illustrated in the code cell below (using dot notation to access functions and constants of the `math` module):

[ ]:
```
import math
x=2*math.pi
f1=math.sinh(x)
f2=0.5*(math.exp(x)-math.exp(-x))
f3=0.5*(math.e**x-math.e**(-x))
print(f"sinh(x)={f1:.4f}, exp(x)-exp(-x)={f2:.4f}, e**x-e**(-x)={f3:.4f}")
```

The three approaches return identical result.

4.2.1 Rounding errors

Our previous example computes a function in three mathematically equivalent ways. Besides, the output from the print statement shows that the three resulting numbers are identical. However, if we print 16 decimal places of `f1`, `f2`, `f3`:

[ ]:
```
print(f"sinh(x)={f1:.16f}, exp(x)-exp(-x)={f2:.16f}, e**x-e**(-x)={f3:.16f}")
```

now `f1` and `f2` are equal, but `f3` differs in the last four decimal places. (This does not mean that `f1` and `f2` are correct, but `f3` doesn't, the three numbers are indeed inaccurate.) Why that happens?

Real numbers need in general an infinite number of digits to be represented exactly. However, **in the computer, storage is finite**. Thus, this infinite sequence of digits has to be truncated at some point. How these numbers are represented is quite technical, and out of the scope of the unit. However, if you are curious, and willing to know more, we recommend this excellent resource.

Another take away from the example above is that one **NEVER** should use the equality operator, i.e., `==` in order to compare if two real numbers are equivalent in the computer. Instead, one checks whether the difference in absolute value of the two numbers is smaller than a tolerance, as shown in the following code cells:

[ ]:
```
a=0.1
b=0.2
expected=0.3
```

```
computed=a+b
tol=1.0e-15
```

[ ]:
```
expected == computed
```

[ ]:
```
abs(expected-computed) < tol
```

4.3 User input. Reading data from the keyboard

Consider the following program for evaluating the formula $y = A\ \sin(\mathrm{wx})$

[ ]:
```
from math import sin
A = 0.1
w = 1
x = 0.6
y = A*sin(w*x)
print(f"{A}*sin({w}*{x})={y}")
```

In this program, the variables `A`, `w`, and `x` can be considered as input data. Their values must be known before the program can perform the computation of `y`. The results produced by the program, `y` in the example at hand, constitute the output data. Input data can be hard-coded in the program as we do above. That is, we explicitly create variables to specific values, `A=0.1`, `w=1`, and `x=0.6`. This programming style can be suitable for small programs, like the ones that we have written so far.

However, in general, **it is better to write programs that let the user provide input data already when the program is running**. As a result, there is no need to modify the program itself when a new set of input data is to be explored. This is an important feature of a program, because modification of the source code always represents a danger of introducing new errors by accident.

The most common ways of reading data data into a program are: 1. The user is asked questions, and provides input data typing it with the keyboard. 2. The user provides input data using the command-line in a Terminal window (e.g., the `git bash` terminal window in Windows, or the Terminal app in MacOS X). For example, assume that we store our program above in a file, say, `formula.py`. When calling the python interpreter from the command-line (and assuming that we have inserted appropriate code into our program, not shown in the cell above), we use `python formula.py --A 0.1 --w 1 --x 0.6`. 3. The user provides input data in a file and the program reads such a file. 4. The user uses a graphical interface (a window) to input data.

In this document we will restrict ourselves to the first way above, as it is indeed the most basic way of getting input data from the user.

Python provides a built-in function, called `input`, that stops the program and waits for the user to type something. When the user presses the `Enter` key, the program resumes and **`input` returns what the user typed as a string (i.e., an object of type `str`)**. `input` gets as an argument a message (a string) that it is wise to use in order to tell the user what the program expects he/she to type.

The usage of `input` is illustrated in the following code cell:

[ ]:
```
name=input("What is your name? (type it with the keyboard and press Enter)")
print(f"Your name is {name}")
```

In the case of the program at the beginning of the section, we have to ask the user for the values of the `A`, `w` and `x` parameters. But recall that `input` always returns an object of type `str`, so that we have to convert it to a floating number using the `float` function:

[ ]:
```
from math import sin

# Messages
what_is_message="What is the value of "
help_message="(Type it and press Enter. Float number expected)"
message_a=what_is_message + "A? " + help_message
message_w=what_is_message + "w? " + help_message
message_x=what_is_message + "x? " + help_message

# Read input from keyboard
A = input(message_a)
A = float(A)
w = input(message_w)
w = float(w)
x = input(message_x)
x = float(x)

# Perform computation
y = A*sin(w*x)

# Print output on screen
print(f"{A}*sin({w}*{x})={y}")
```

5 How to find more Python information

The Jupyter notebooks that are delivered to you in this course only contain fragments of the Python language. When doing the tasks and exercises you will certainly need for looking up more detailed information on modules, objects, etc. Fortunately, there is a lot of excellent documentation on-line.

The primary reference is the official Python documentation website: docs.python.org. Here you can find a Python tutorial, the very useful Library Reference (`https://docs.python.org/3/library/`), and a Language Reference, to mention some key documents.

Alternatively, you can ask for help at any time calling `help(concept)`, e.g., `help(math)`, either on the Python interpreter or Jupyter notebook. This is illustrated in the cell below.

[ ]:
```
import math
help(math)
```

However, **an important caveat comes here**. For a novel programmer it is difficult to read manuals (they are mostly intended for experts!). Better browse, don't read everything, try to dig out the key info. It's much like googling in general: only a fraction of the information is relevant for you.