In [1]:
```
import pybryt
from lecture import pybryt_reference
```

Introduction to Python

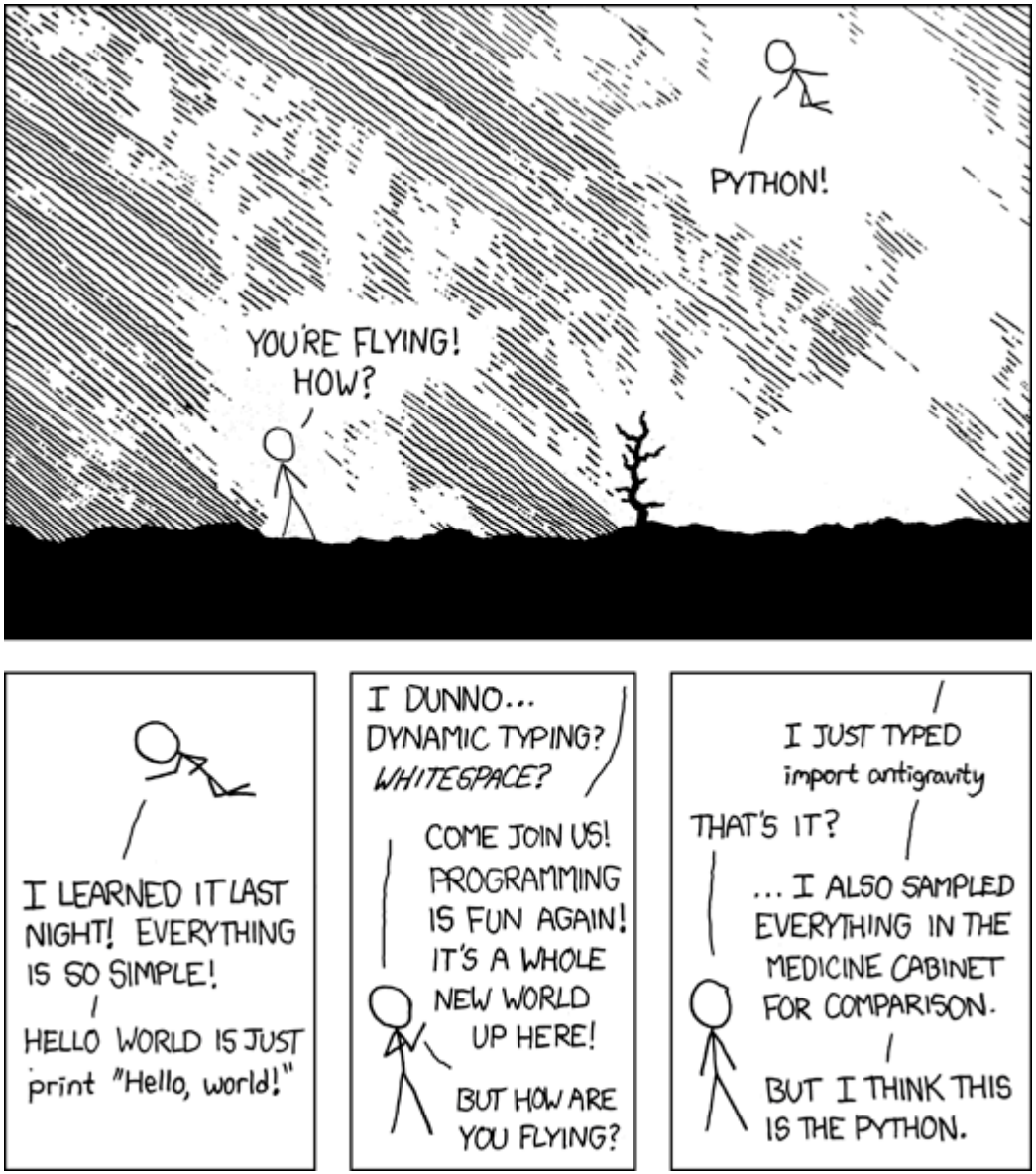
Lecture 1: Hello world, conditional expressions, loops, and lists

Learning objectives:

At the end of this lecture, you will:

- Know that Python will help you defy gravity.
- Execute Python *statements* from within Jupyter.
- Explain what a *variable* is and express a mathematical formula in code.
- Print program outputs.
- Access mathematical functions from a Python module.
- Write your own *function*.
- Form a *condition* using a *boolean expression*.
- Use a conditional expression in combination with a `while`-loop to perform repetitive tasks.
- Store data elements within a Python `list`.
- Use a `for`-loop to iterate, and perform some task over a sequence of elements.

```
import antigravity
```



Programming a mathematical formula

Here is a formula for the position of a ball, $y(t)$, in vertical motion, starting at ground level (i.e. at $y=0$) at time $t=0$:

$$ y(t) = v_{0}t- \frac{1}{2}gt^2, $$

where:

- $y(t)$ is the height (position) as a function of time t,
- v_0 is the initial velocity (at $t=0$), and
- g is the acceleration due to gravity.

The computational task we want to solve is: given the values of v_0, g, and t, compute the height y.

How do we program this task? A program is a sequence of instructions given to the computer. However, while a programming language is much **simpler** than a natural language, it is more **pedantic**. Programs must have correct syntax, i.e. correct use of the computer language grammar rules, and no misprints.

So let us execute a Python statement based on this example to evaluate $y(t) = v_0t- \frac{1}{2}gt^2$ for $v_0 = 5 \,\text{ms}^{-1}$, $g = 9.81 \,\text{ms}^{-2}$ and $t = 0.6 \,\text{s}$. If you were doing this on paper, you would probably write something like this: $$y = 5\cdot 0.6 - \frac{1}{2}\cdot 9.81 \cdot 0.6^2.$$ Happily, writing this in Python is very similar:

In [2]:
```
print(5 * 0.6 - 0.5 * 9.81 * 0.6**2)
```

```
1.2342
```

You probably noticed that, in the code cell we just wrote, the first few lines start with the hash (`#`) character. In Python, we use `#` to tell the Python interpreter to ignore everything that comes after `#` . We call those lines **comments**, and we write them to help us (humans) understand the code. Besides, we used `print` and we enclosed our calculation within parentheses to display the output. We will explain *comments* and *print function* in more detail later.

Exercise 1.1: Open a code cell and write some code.

1. Navigate the Jupyter toolbar to "Insert"->"Insert Cell Below". Note from the toolbar that you can select a cell to be 'Code' (this is the default), 'Markdown' (the cell you are reading right now is written in Markdown - double click this cell to investigate further), and 'Raw' (its content is not evaluated by the notebook).
2. Copy&paste the code from the previous code cell into your newly created code cell below. Make sure it runs!

3. To see how important it is to use the correct syntax, replace `**` with `^` in your code and try running the cell again. You should see something like the following:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-6e110a567f02> in <module>
      3 # cell, or click on the 'Run' widget on the Jupyter toolbar above.
      4
----> 5 print(5*0.6 - 0.5*9.81*0.6^2)

TypeError: unsupported operand type(s) for ^: 'float' and 'int'
```

4. Undo that change so your code is working again; now change `print` to `write` and see what happens when you run the cell. You should see something like:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-a3da902ceb19> in <module>
      3 # cell, or click on the 'Run' widget on the Jupyter toolbar above.
      4
----> 5 write(5*0.6 - 0.5*9.81*0.6**2)

NameError: name 'write' is not defined
```

While a human might still understand these statements, they do not mean anything to the Python interpreter. Rather than throwing your hands up in the air whenever you get an error message like the above (you are going to see many during your course), train yourself to read error messages carefully to get an idea what it is complaining about, and re-read your code from the perspective of the Python interpreter.

Error messages can look bewildering and even frustrating at first, but **it gets much easier with practise**.

In [3]:
```
print(5 * 0.6 - 0.5 * 9.81 * 0.6**2)
```

```
1.2342
```

In [4]:
```
print(5*0.6 - 0.5*9.81*0.6^2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(5*0.6 - 0.5*9.81*0.6^2)

TypeError: unsupported operand type(s) for ^: 'float' and 'int'
```

In [5]:
```
write(5*0.6 - 0.5*9.81*0.6**2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 write(5*0.6 - 0.5*9.81*0.6**2)

NameError: name 'write' is not defined
```

Storing numbers in variables

From mathematics, you are already familiar with *variables* (e.g. $v_0 = 5$, $g = 9.81$, $t = 0.6$, $y = v_0t - \frac{1}{2}gt^2$), and know how important they are for working out complicated problems. Similarly, you can use variables in a program to make it easier to read and understand.

In [6]:
```
v0 = 5
g = 9.81
t = 0.6
y = v0 * t - 0.5 * g * t**2
print(y)
```

```
1.2342
```

This program performs the same calculations as the previous one and gives the same output. However, this program spans several lines and uses variables.

We usually use one letter for a variable in mathematics, resorting to using the Greek alphabet and other characters for more clarity. The main reason for this is to avoid becoming exhausted from writing when working out long expressions or derivations. However, when programming, you should use more descriptive names for variables. This might not seem like an important consideration for

the trivial example here. Still, it becomes increasingly important as the program gets more complicated and if someone else has to read your code.

Good variable names make a program easier to understand!

Python *allows* variable names to include:

- lowercase `a-z`, uppercase `A-Z`, underscore (`_`), and digits `0-9`, **but** the name cannot start with a digit or contain a whitespace.
- "textlike" Unicode characters from other languages are also allowed.

Variable names are case-sensitive (strictly, character sensitive, i.e. `a` is different from `A`).

Good variable names are often:

- standard one-letter symbols,
- words or abbreviations of words,
- phrases joined together in snake_case or camelCase.

Let us rewrite the previous example using more descriptive variable names:

In [7]:
```
initial_velocity = 5
acceleration_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = (
    initial_velocity * TIME - 0.5 * acceleration_of_gravity * TIME**2
)
print(VerticalPositionOfBall)
```

```
1.2342
```

In Python, you can check if the name you would like to give to your variable is valid or not, i.e. check if it is a valid identifier. To do that, you can enclose the variable name in quotes (to form a string literal) and call its `isidentifier()` method:

In [8]:
```
print("initial_velocity".isidentifier())  # snake_case-style variable name
print("InitialVelocity".isidentifier())  # CamelCase-style variable name
```

```
print("initial velocity".isidentifier())  # variable name contains space
```

```
True
True
False
```

Certain words have a **special meaning** in Python and **cannot be used as variable names**. We refer to them as **keywords**, and in Python 3.9 they are:

`and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, and `yield`.

Similarly, `True`, `False`, and `None` are keywords we use for the values of variables, and we cannot use them for variable names. Keywords are very important in programming and, in these lectures, we will learn how to use some of them.

Finally, Python has some "builtin" functions which are always available. While Python will let you do it, it is usually a bad idea to use these names, since this would *overshadow* the builtin function and prevent us from calling it. The full list of builtin functions in Python 3.9 is `abs()`, `aiter()`, `all()`, `any()`, `anext()`, `ascii()`, `bin()`, `bool()`, `breakpoint()`, `bytearray()`, `bytes()`, `callable()`, `chr()`, `classmethod()`, `compile()`, `complex()`, `delattr()`, `dict()`, `dir()`, `divmod()`, `enumerate()`, `eval()`, `exec()`, `filter()`, `float()`, `format()`, `frozenset()`, `getattr()`, `globals()`, `hasattr()`, `hash()`, `help()`, `hex()`, `id()`, `input()`, `int()`, `isinstance()`, `issubclass()`, `iter()`, `len()`, `list()`, `locals()`, `map()`, `max()`, `memoryview()`, `min()`, `next()`, `object()`, `oct()`, `open()`, `ord()`, `pow()`, `print()`, `property()`, `range()`, `repr()`, `reversed()`, `round()`, `set()`, `setattr()`, `slice()`, `sorted()`, `staticmethod()`, `str()`, `sum()`, `super()`, `tuple()`, `type()`, `vars()`, and `zip()`.

Please note that the list of keywords and builtins may differ between different Python versions.

Adding comments to code

Not everything written in a computer program is intended for execution. In Python, anything on a line after the `#` character is ignored and is known as a **comment**. You can write whatever you want in a comment. Comments are intended to be used to explain what a snippet of code is intended for. It might, for example, explain the objective or provide a reference to the data or algorithm used. This is useful for you when you have to understand your code at some later stage, and indeed for whoever has to read and understand your code later.

In [9]:
```
# Program for computing the height of a ball in vertical motion.
v0 = 5  # Set initial velocity in m/s.
g = 9.81  # Set acceleration due to gravity in m/s^2.
t = 0.6  # Time at which we want to know the height of the ball in seconds.
y = v0 * t - 0.5 * g * t**2  # Calculate the vertical position.
print(y)
```

```
1.2342
```

Exercise 1.2: Convert from metres to British length units

Here in the UK, we are famous for our love of performing mental arithmetic. That is why we still use both imperial and metric measurement systems - hours of fun entertainment for the family switching back and forth between the two.

Make a program where you set a length given in metres and then compute and write out the corresponding length measured in:

- inches (one inch is 2.54 cm)
- feet (one foot is 12 inches)
- yards (one yard is 3 feet)
- miles (one British mile is 1760 yards)

Note: In this course, we are using pybryt for automated assessment scoring. Therefore, while it is generally important to always carefully follow the instructions of a question, it is particularly important here so that *pybryt* can recognise the validity of your answer. In addition to PyBryt, after each exercise, there is a cell with `assert` statements for additional testing in case your solution is not covered with PyBryt references.

Uncomment and modify the relevant lines in the following code cell. The conversion to inches is done for you to illustrate what is required.

In [10]:
```
metres = 640

# 1 inch = 2.54 cm. Remember to convert 2.54 cm to 0.0254 m here.
inches = metres / 0.0254

# Uncomment and complete the following code. Do not change variable names for testing.
feet = inches / 12
yards = feet / 3
miles = yards / 1760
```

In [11]:
```
with pybryt.check(pybryt_reference(1, 2)):
    feet, yards, miles
```

```
REFERENCE: exercise-1_2
SATISFIED: True
MESSAGES:
  - SUCCESS: Your conversion to feet is correct. Well done!
  - SUCCESS: Amazing! Your calculation of yards is correct.
  - SUCCESS: Wow! Your conversion to miles is right.
```

In [12]:
```
import numbers
import numpy as np

assert all([isinstance(i, numbers.Real) for i in [feet, yards, miles]])

### BEGIN HIDDEN TESTS
assert np.isclose(feet, 2099.737532808399)
assert np.isclose(yards, 699.912510936133)
assert np.isclose(miles, 0.3976775630318938)
### END HIDDEN TESTS
```

Formatted printing style

Often we want to print out results using a combination of text and numbers, e.g. `"At t=0.6 s, y is 1.23 m"`. In Python, we can do this using f-strings:

In [13]:
```
print(f"At t={t} s, y is {y} m.")  # f-string method - string literal begins with an f
```

```
At t=0.6 s, y is 1.2342 m.
```

We enclose our sentence in (single or double) quotes to denote a string literal and add `f` in front of it to tell Python to replace `{t}` and `{y}` with the values of `t` and `y`, respectively.

When printing out floating-point numbers, we should **never** quote numbers to a higher accuracy than they were measured. Python provides a *printf formatting* syntax exactly for this purpose. We can see in the following example where the *format* `g` expresses the floating-point number with the minimum number of significant figures, and the *format* `.2f` specifies that only two digits are printed out after the decimal point. We specify the format inside `{}` and separate it from the variable name with `:`.

In [14]:
```
print(f"At t={t:g} s, y is {y:.2f} m.")  # f-string with specified formatting
```

```
At t=0.6 s, y is 1.23 m.
```

Sometimes we want a multi-line output. This is achieved using a triple quotation, i.e. `"""`:

In [15]:
```
print(f"""At t={t:f} s, a ball with
initial velocity v0={v0:.3E} m/s
is located at the height y={y:.2f} m.
""")
```

```
At t=0.600000 s, a ball with
initial velocity v0=5.000E+00 m/s
is located at the height y=1.23 m.
```

Notice in this example we used `f`, `.3E`, and `.2f` to define formats, into which we inserted the values of `t`, `v0`, and `y` respectively. You can find more details about the format specification mini-language in the Python documentation.

Instead of using the f-string formatted printing, Python offers another two syntax alternatives: string's `format` method and the `%` operator. Let us have a look at how we can print `"At t=0.6 s, y is 1.23 m"` using these two alternative solutions.

In [16]:
```
print("At t={:g} s, y is {:.2f} m.".format(t, y))  # string's format method
print("At t=%g s, y is %.2f m." % (t, y))  # % operator
```

```
At t=0.6 s, y is 1.23 m.
At t=0.6 s, y is 1.23 m.
```

Notice that we defined slots in a string using curly braces `{}` where we also specified the formatting style in the same way we did before. We inserted the values into the slots by passing them to the `format()` method or by writing them in curly braces.

The `%` operator expands out the input tuple place by place (so the first *slot* gets the first element, the second the second, and so on). If there is only one *slot* then using a tuple is optional.

Exercise 1.3: Compute the air resistance on a football

The drag force, due to air resistance, on an object can be expressed as $$F_d = \frac{1}{2}C_D\rho Av^2$$ where:

- ρ is the density of the air,
- v is the velocity of the object,
- A is the cross-sectional area (perpendicular to the velocity direction),
- and C_D is the drag coefficient, which depends on the shape of the object and the roughness of the surface.

Complete and fix the following code that computes the drag force.

In [17]:
```
density = 1.2  # units of kg/m^3
ball_radius = 0.11  # units of m
C_D = 0.2  # drag coefficient
v = 50.8  # m/s (fastest recorded speed of football)

from math import pi
# Uncomment, fix, and complete the following lines.
A = pi*ball_radius**2  # cross sectional area of a sphere
F_d = 1/2 * C_D * density * A * v**2

# Challenge yourself to use the formatted print statement
# shown above to write out the force with one decimal in
# units of Newton (1 N = 1 kgm/s^2).
print(f"The drag force is {F_d}")
```

```
The drag force is 11.771828154393067
```

In [18]:
```
with pybryt.check(pybryt_reference(1, 3)):
    A, F_d
```

```
REFERENCE: exercise-1_3
SATISFIED: True
MESSAGES:
  - SUCCESS: Computed cross-sectional area is correct. Great!
  - SUCCESS: Your calculation of the drag force is correct.
```

In [19]:
```
import numbers
import numpy as np

assert isinstance(F_d, numbers.Real)

### BEGIN HIDDEN TESTS
assert isinstance(A, numbers.Real)
assert np.isclose(F_d, 11.771828154393067)
assert np.isclose(A, 0.0380132711084365)
### END HIDDEN TESTS
```

How are arithmetic expressions evaluated?

Consider the random mathematical expression:

$$ \frac{5}{9} + \frac{3a^4}{2} $$

implemented in Python as `5/9 + 3 * a**4 / 2`. The rules for evaluating the expression are the same as in mathematics: proceed term by term (additions/subtractions) from the left, compute powers first, then multiplication and division. Therefore in this example the order of evaluation will be:

1. `r1 = 5/9`
2. `r2 = a**4`
3. `r3 = 3*r2`
4. `r4 = r3/2`
5. `r5 = r1 + r4`

We use parenthesis to override these default rules. Indeed, many programmers use parenthesis for greater clarity.

Exercise 1.4: Compute the growth of money in a bank

Let p be a bank's annual interest rate in per cent. After n years, an initial amount A_0 has then grown to

$$A_n = A_0\left(1+\frac{p}{100}\right)^n.$$

Write a program for computing how much money 1000 euros have grown to after three years with a 5% annual interest rate.

In [20]:
```
# Uncomment and complete the code below (and, as always, don't change variable names)
p = 5
A_0 = 1000
n = 3
A_n = A_0 * (1 + p / 100)**n
print(f"The amount of money in the account after {n:d} years is {A_n:.2f} euros")
```

```
The amount of money in the account after 3 years is 1157.63 euros
```

In [21]:
```
with pybryt.check(pybryt_reference(1, 4)):
    p, A_0, n, A_n
```

```
REFERENCE: exercise-1_4
SATISFIED: True
MESSAGES:
  - SUCCESS: You set the value of the annual interest rate (p) correctly.
  - SUCCESS: You set the value of the initial amount (A_0) correctly.
  - SUCCESS: You set the number of years (n) correctly.
  - SUCCESS: Amazing! Your final solution (A_n) is correct.
```

In [22]:
```
import numbers
import numpy as np

assert isinstance(A_n, numbers.Real)

### BEGIN HIDDEN TESTS
assert np.isclose(A_n, 1157.6250000000002)
### END HIDDEN TESTS
```

Standard mathematical functions

What if we need to compute $\sin x$, $\cos x$, $\ln x$, e^x etc., in a program? Such functions are available in Python's `math` module. In fact, there is a vast universe of functionality for Python available in modules. We just `import` in whatever we need for the task at hand.

In this example, we compute $\sqrt{2}$ using the `sqrt` function from the `math` module:

In [23]:
```
import math

# Since we imported library (import math),
# we access the sqrt function using math.sqrt.
r = math.sqrt(2)
print(r)
```

```
1.4142135623730951
```

or:

In [24]:
```
from math import sqrt

# This time, we did not import the entire library -
# we imported only sqrt function.
# Therefore, we can use it directly.
r = sqrt(2)
print(r)
```

```
1.4142135623730951
```

Let us now have a look at a more complicated expression, such as

$$\sin x \cos x + 4\ln x$$

In [25]:
```
from math import sin, cos, log

x = 1.2
print(sin(x) * cos(x) + 4 * log(x))  # log is ln (base e)
```

```
1.0670178174513938
```

Exercise 1.5: Evaluate a Gaussian function

The bell-shaped Gaussian function,

$$f(x)=\frac{1}{s\sqrt{2\pi}}\exp\left(-\frac{1}{2} \left(\frac{x-m}{s}\right)^2\right)$$

is one of the most widely used functions in science and engineering. The mean m and standard deviation s are real numbers, and s must be greater than zero. Write a program for evaluating the Gaussian function when $m = 0$, $s = 2$, and $x = 1$.

In [26]:
```
# Uncomment and complete the code below (don't change variable names)
from math import pi, exp, sqrt
m, s, x = 0, 2, 1
f_x = 1 / (s * sqrt(2 * pi)) * exp( -1 / 2 * ((x - m) / s)**2)
print(f"f_x is {f_x}")
```

```
f_x is 0.17603266338214976
```

In [27]:
```
with pybryt.check(pybryt_reference(1, 5)):
    f_x
```

```
REFERENCE: exercise-1_5
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You computed the value of the Gaussian function correctly.
```

In [28]:
```
import numbers
import numpy as np

assert isinstance(f_x, numbers.Real)

### BEGIN HIDDEN TESTS
assert np.isclose(f_x, 0.17603266338214976)
### END HIDDEN TESTS
```

Exercise 1.6: Find and fix errors in the coding of a formula

Roots of a quadratic equation $ax^2 + bx + c = 0$ are:

$$x_1 = \frac{-b + \sqrt{b^2 -4ac}}{2a},$$ and $$x_2 = \frac{-b - \sqrt{b^2 -4ac}}{2a}.$$

Uncomment and fix the errors in the following code.

In [29]:
```
from math import sqrt
a = 2
b = 1
c = -2
q = sqrt(b*b - 4*a*c)
x1 = (-b + q) / (2 * a)
x2 = (-b - q) / (2 * a)
print(f"q is {q}, x1 is {x1}, x2 is {x2}")
```

```
q is 4.123105625617661, x1 is 0.7807764064044151, x2 is -1.2807764064044151
```

In [30]:
```
with pybryt.check(pybryt_reference(1, 6)):
    q, x1, x2
```

```
REFERENCE: exercise-1_6
SATISFIED: True
MESSAGES:
  - SUCCESS: You compute the square root of the discriminant (q) correctly.
  - SUCCESS: Root x1 is correct.
  - SUCCESS: Root x2 is correct.
```

In [31]:
```
import numbers
import numpy as np

assert isinstance(q, numbers.Real)
assert isinstance(x1, numbers.Real)
assert isinstance(x2, numbers.Real)

### BEGIN HIDDEN TESTS
assert np.isclose(q, 4.123105625617661)
assert np.isclose(x1, 0.7807764064044151)
assert np.isclose(x2, -1.2807764064044151)
### END HIDDEN TESTS
```

Functions

We have already used Python functions above, e.g. `sqrt` from the `math` module. In general, a function is a collection of statements we can execute wherever and whenever we want. For example, consider any of the formulae we implemented above.

Functions can take any number of inputs (called *arguments* or *parameters*) to produce outputs. Functions help to organise programs, make them more understandable, shorter, and easier to extend. Wouldn't it be nice to implement it just once and then be able to use it again any time you need it, rather than having to write out the whole formula again?

For our first example, we will reuse the formula for the position of a ball in a vertical motion, which we have seen earlier.

In [32]:
```python
def ball_height(v0, t, g=9.81):
    """Function to calculate and return height of the ball in vertical motion.

    Parameters
    ----------
    v0 : float
        Initial velocity (units, m/s).
    t : float
        Time at which we want to know the height of the ball (units, seconds).
    g : float, optional
        Acceleration due to gravity (units, m/s^2). By default 9.81 m/s^2.

    Returns
    -------
    float
        Height of the ball in metres.

    """
    height = v0 * t - 0.5 * g * t**2

    return height
```

Let us break this example down:

- Function *signature* (header):

- Functions start with `def` followed by the name we want to give the function (`ball_height` in this case). Just like with variables, function names must be valid identifiers.
- Following the name, we have parentheses followed by a colon `(...):` containing zero or more function *arguments*.
- In this case, `v0` and `t` are *positional arguments*, while `g` is known as a *keyword argument* (more about this later).

- Function *body*:
  - The first thing to notice is that the body of the function is indented one level. All code that is indented with respect to `def` -line belongs to a function.
  - Best practice is to include a docstring to explain to others (or remind our future self) how to use the function. Docstring is defined as a multi-line string literal (enclosed in triple quotes """).
  - The function output is passed back via the `return` statement.

Notice that this just defines the function. Nothing is executed until we actually *call* the function:

In [33]:
```
# We pass 5 and 0.6 to v0 and t, respectively.
# Since we do not pass a value for g,
# a default value we defined in function's signature is used.
# The value function returns (height) is put in variable h.
h = ball_height(5, 0.6)

print(f"Ball height: {h:g} metres.")
```

```
Ball height: 1.2342 metres.
```

No return value implies that `None` is returned. `None` is a special Python object (singleton) that semantically often represents an "empty" or undefined value. It is surprisingly useful, and we will use it a lot later.

Functions can also return multiple values. Let us extend the previous example to calculate the ball's velocity as well as its height:

In [34]:
```
def ball_height_velocity(v0, t, g=9.81):
    """Function to calculate ball's height and its velocity.

    Parameters
    ----------
    v0 : float
        Initial velocity (units, m/s).
    t : float
        Time at which we want to know the height of the ball (units, seconds).
```

```
    g : float, optional
        Acceleration due to gravity (units, m/s^2). By default 9.81 m/s^2.

    Returns
    -------
    float
        Height of ball in metres.
    float
        Velocity of ball in m/s.

    """
    height = v0 * t - 0.5 * g * t**2
    velocity = v0 - g * t

    return height, velocity


# We pass 5 and 0.6 to v0 and t, respectively.
# The first value function returns (height) is put into variable h,
# whereas the second one (velocity) is placed in v - iterable unpacking.
h, v = ball_height_velocity(5, 0.6)

print("Ball height: %g metres." % h)
print("Ball velocity: %g m/s." % v)
```

```
Ball height: 1.2342 metres.
Ball velocity: -0.886 m/s.
```

Scope: Local and global variables

Variables defined within a function are said to have *local scope*. That is to say that we can only reference them within that function. Consider the example function defined above where we used the *local* variable *height*. You can see that if you try to print the variable height outside the function, you will get an error.

```
print(height)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-50-aa6406a13920> in <module>
```

```
----> 1 print(height)

NameError: name 'height' is not defined
```

Keyword arguments and default input values

Besides *positional arguments*, functions can have arguments of the form `argument_name=value` and they are called *keyword arguments*:

In [35]:
```
def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
    print(f"arg1: {arg1}, arg2: {arg2}, kwarg1: {kwarg1}, kwarg2: {kwarg2}")


# Note that we have not specified inputs for kwarg1 and kwarg2.
somefunc("Hello", [1, 2])
```

```
arg1: Hello, arg2: [1, 2], kwarg1: True, kwarg2: 0
```

In [36]:
```
# Note that we replace the default value for kwarg1.
somefunc("Hello", [1, 2], kwarg1="Hi")
```

```
arg1: Hello, arg2: [1, 2], kwarg1: Hi, kwarg2: 0
```

In [37]:
```
# Note that we replace the default value for kwarg2.
somefunc("Hello", [1, 2], kwarg2="Hi")
```

```
arg1: Hello, arg2: [1, 2], kwarg1: True, kwarg2: Hi
```

In [38]:
```
# Here, we replace both default values for keyword arguments kwarg1 and kwarg2.
somefunc("Hello", [1, 2], kwarg2="Hi", kwarg1=6)
```

```
arg1: Hello, arg2: [1, 2], kwarg1: 6, kwarg2: Hi
```

If we use `argument_name=value` for all arguments, their sequence in the function call can be in any order.

In [39]:
```
somefunc(kwarg2="Hello", arg1="Hi", kwarg1=6, arg2=[2])
```

```
arg1: Hi, arg2: [2], kwarg1: 6, kwarg2: Hello
```

Exercise 1.7: Implement a Gaussian function

Write a Python function `gaussian` to compute the Gaussian function:

$$f(x)=\frac{1}{s\sqrt{2\pi}}\exp\left(-\frac{1}{2} \left(\frac{x-m}{s}\right)^2\right)$$

In [40]:
```
# Uncomment and complete this code - keep the function name the same for testing purposes.
def gaussian(x, m=0, s=1):
    f_x = 1 / (s * sqrt(2 * pi)) * exp( -1 / 2 * ((x - m) / s)**2)
    return f_x
f_x = gaussian(x=2)
print(f_x)
```

```
0.05399096651318806
```

In [41]:
```
with pybryt.check(pybryt_reference(1, 7)):
    gaussian(0.5)
```

```
REFERENCE: exercise-1_7
SATISFIED: True
MESSAGES:
  - SUCCESS: Your implementation of the Gaussian function is correct. Well done!
```

In [42]:
```
import numbers
import numpy as np

res = gaussian(x=0, m=0, s=1)
assert isinstance(res, numbers.Real)
assert np.isclose(res, 1 / np.sqrt(2 * np.pi))

### BEGIN HIDDEN TESTS
assert callable(gaussian)
assert np.isclose(gaussian(x=2, m=0, s=1), 0.05399096651318806)
### END HIDDEN TESTS
```

Exercise 1.8: How to cook the perfect egg

You just started University and moved away from home. You're trying to impress your new flatmates by cooking brunch. Write a Python script to help you cook the perfect egg!

You know from A-levels that, when the temperature exceeds a critical point, the proteins in the egg first denature and then coagulate. The process becomes faster as the temperature increases. In the egg white, the proteins start to coagulate for temperatures above 63°C, while in the yolk the proteins start to coagulate for temperatures above 70°C.

The time t (in seconds) it takes for the centre of the yolk to reach the temperature T_y (in degrees Celsius) can be expressed as:

$$t = \frac{M^{2/3}c \rho^{1/3}}{K \pi^2 (4 \pi /3)^{2/3} } ln \left[0.76 \frac{T_0-T_w}{T_y-T_w}\right]$$

where:

- M is the mass of the egg;
- ρ is the density;
- c is the specific heat capacity;
- K is thermal conductivity;
- T_w temperature of the boiling water (in $^\circ$C);
- T_0 is the initial temeprature of the egg (in $^\circ$C), before being put in the water.

Write a function that returns the time t needed for the egg to cook, knowing that $T_w = 100^\circ\text{C}$, $M = 50\,\text{g}$, $\rho = 1.038\,\text{gcm}^{-3}$, $c = 3.7\,\text{Jg}^{-1}\text{K}^{-1}$, and $K = 5.4 \cdot 10^{-3}\,\text{Wcm}^{-1}\text{K}^{-1}$. Find t for an egg taken from the fridge ($T_0 = 4^\circ\text{C}$) and for one at room temperature ($T_0 = 20^\circ\text{C}$). $T_y = 70^\circ\text{C}$ for a perfect soft-boiled egg.

Hint: You do not need to do any unit conversion.

In [43]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.

from math import pi, log

def perfect_egg(T0, M=50, rho=1.038, Tw=100, c=3.7, K=5.4e-3, Ty=70):
    numerator = M**(2/3) * c * rho**(1/3)
    denominator = K * (pi**2) * ((4*pi/3)**(2/3))
```

```
    t = (numerator / denominator) * log(0.76 * ((T0 - Tw) / (Ty - Tw)))
    return t
res1 = perfect_egg(T0=4)
res2 = perfect_egg(T0=20)
print(f"res1 is {res1}\nres2 is {res2}")
```

```
res1 is 326.2798626986453
res2 is 259.3428560570137
```

In [44]:
```
with pybryt.check(pybryt_reference(1, 8)):
    perfect_egg(T0=4, M=50, rho=1.038, Tw=100, c=3.7, K=5.4e-3, Ty=70)
    perfect_egg(T0=20, M=50, rho=1.038, Tw=100, c=3.7, K=5.4e-3, Ty=70)
```

```
REFERENCE: exercise-1_8
SATISFIED: True
MESSAGES:
  - SUCCESS: Your function computes correct time for an egg at the fridge temperature to be perfectly cooked. Well done!
  - SUCCESS: The time for an egg at the room temperature to be perfectly cooked is correct. Well done!
```

In [45]:
```
import numbers
import numpy as np

res1 = perfect_egg(T0=4, M=50, rho=1.038, Tw=100, c=3.7, K=5.4e-3, Ty=70)
res2 = perfect_egg(T0=20, M=50, rho=1.038, Tw=100, c=3.7, K=5.4e-3, Ty=70)
assert isinstance(res1, numbers.Real)
assert isinstance(res2, numbers.Real)

### BEGIN HIDDEN TESTS
assert callable(perfect_egg)
assert np.isclose(res1, 326.2798626986453)
assert np.isclose(res2, 259.3428560570137)
### END HIDDEN TESTS
```

Exercise 1.9: Kepler's third law

You were selected to be the next astronaut to go to Mars. Congratulations!

Kepler's third law expresses the relationship between the distance of planets from the Sun, a, and their orbital periods, P:

$$ P^2 = \frac{4\pi^2}{G(M + m)}a^3 $$

where

- P is the period (in seconds);
- G is the gravitational constant ($G = 6.67 \cdot 10^{-11} \,\text{m}^3\text{kg}^{-1}\text{s}^{-2}$);
- M is the mass of the Sun ($M = 2 \cdot 10^{30} \text{kg}$);
- m is the mass of the planet (in kg);
- a is the distance between the planet and the Sun (in m).

How many Earth birthdays will you celebrate during your 10-years Mars mission? Write a Python function `period` that calculates the period of a planet. Using `period` function, calculate the period of the Earth, `P_earth`, and the period of Mars, `P_mars`. Finally, calculate `birthdays` which is how many Earth years are equivalent to 10 years on Mars.

We know that:

- The average distance between the Earth and the Sun is $a = 1.5 \cdot 10^{11} \,\text{m}$;
- The average distance between Mars and the Sun is 0.5 larger than the Earth-Sun distance;
- The mass of the Earth is $m_1 = 6 \cdot 10^{24} \,\text{kg}$;
- Mars's mass is about 10% of the Earth's mass.

Hint: You do not need to do any unit conversion.

In [46]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
from math import pi, sqrt

def period(a, m_planet, m_sun=2e30, G=6.67e-11):
    P = sqrt((4 * pi**2 * a**3) / (G * (m_planet + m_sun)))
    return P

a_earth = 1.5e11
a_mars = 1.5 * a_earth
m_earth = 6e24
m_mars = 0.1 * m_earth
P_mars = period(a_mars, m_mars)
P_earth = period(a_earth, m_earth)
```

```
birthdays = 10 * P_mars /P_earth

print(P_earth)
print(P_mars)
print(birthdays)
```

```
31603718.929927427
58059817.3950661
18.3711978719333
```

In [47]:
```
with pybryt.check(pybryt_reference(1, 9)):
    period(a=1e11, m_planet=1e24), P_mars, P_earth, birthdays
```

```
REFERENCE: exercise-1_9
SATISFIED: True
MESSAGES:
  - SUCCESS: Your function computes the period correctly.
  - SUCCESS: You computed the Earth's period correctly.
  - SUCCESS: You computed Mars's period correctly.
  - SUCCESS: Wow! Your computed birthdays value is correct!
```

In [48]:
```
import numbers
import numpy as np

res = period(a=1, m_planet=1, m_sun=1, G=0.5)
assert isinstance(res, numbers.Real)
assert np.isclose(res, 2 * np.pi)

### BEGIN HIDDEN TESTS
assert callable(period)
assert np.isclose(P_earth, 31603718.929927427)
assert np.isclose(P_mars, 58059817.3950661)
assert np.isclose(birthdays, 18.3711978719333)
### END HIDDEN TESTS
```

Boolean expressions

An expression with value `True` or `False` is called a boolean expression. Example expressions for what you would write mathematically as $C = 40$, $C \ne 40$, $C \ge 40$, $C \gt 40$ and $C \lt 40$ are:

```
C == 40  # Note: the double == checks for equality!
C != 40  # This could also be written as "not C == 4"
C >= 40
C > 40
C < 40
```

Let us now test some boolean expressions:

In [49]:
```
C = 41

print("C != 40: ", C != 40)
print("C < 40: ", C < 40)
print("C == 41: ", C == 41)
```

```
C != 40:  True
C < 40:  False
C == 41:  True
```

Several conditions can be combined with keywords `and` and `or` into a single boolean expression:

- **Rule 1**: (`C1 and C2`) is `True` only if both `C1` and `C2` are `True`.
- **Rule 2**: (`C1 or C2`) is `True` if either `C1` or `C2` are `True`.

Examples:

In [50]:
```
x = 0
y = 1.2

print("x >= 0 and y < 1:", x >= 0 and y < 1)
print("x >= 0 or y < 1:", x >= 0 or y < 1)
```

```
x >= 0 and y < 1: False
x >= 0 or y < 1: True
```

Exercise 1.10: Values of boolean expressions

Add a comment to the code below to explain the outcome of each of the boolean expressions:

In [51]:
```
C = 41

print("Case 1: ", C == 40)  # False because C is not equal to 40
print("Case 2: ", C != 40 and C < 41)  # False because C is not smaller than 40
print("Case 3: ", C != 40 or C < 41)  # True because C is not equal to 40
print("Case 4: ", not C == 40) # True because C is equal to 41
print("Case 5: ", not C > 40)  # False because C is bigger than 40
print("Case 6: ", C <= 41)  # True because C is equal to 40
print("Case 7: ", not False) # True
print("Case 8: ", True and False) # False
print("Case 9: ", False or True) #True
print("Case 10: ", False or False or False) # False
print("Case 11: ", True and True and False) # False
print("Case 12: ", (True and True) or 1 == 2) # True
```

```
Case 1:  False
Case 2:  False
Case 3:  True
Case 4:  True
Case 5:  False
Case 6:  True
Case 7:  True
Case 8:  False
Case 9:  True
Case 10:  False
Case 11:  False
Case 12:  True
```

Loops

Suppose we want to make the following table of Celsius and Fahrenheit degrees:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  5  41.0
```

```
10  50.0
15  59.0
20  68.0
25  77.0
30  86.0
35  95.0
40 104.0
```

How do we write a program that prints out such a table? We know that $F = \frac{9}{5}C + 32$, and a single line in this table is:

In [52]:
```
C = -20
F = 9 / 5 * C + 32

print(C, F)
```

```
-20 -4.0
```

Now, we can just repeat these statements:

In [53]:
```
C = -20
F = 9 / 5 * C + 32
print(C, F)
C = -15
F = 9 / 5 * C + 32
print(C, F)
C = -10
F = 9 / 5 * C + 32
print(C, F)
C = -5
F = 9 / 5 * C + 32
print(C, F)
C = 0
F = 9 / 5 * C + 32
print(C, F)
C = 5
F = 9 / 5 * C + 32
print(C, F)
C = 10
F = 9 / 5 * C + 32
```

```
print(C, F)
C = 15
F = 9 / 5 * C + 32
print(C, F)
C = 20
F = 9 / 5 * C + 32
print(C, F)
C = 25
F = 9 / 5 * C + 32
print(C, F)
C = 30
F = 9 / 5 * C + 32
print(C, F)
C = 35
F = 9 / 5 * C + 32
print(C, F)
C = 40
F = 9 / 5 * C + 32
print(C, F)
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

We can see that works but it is **very boring** to write and very easy to introduce a misprint.

You really should not be doing boring repetitive tasks like this. Spend your time instead looking for a smarter solution. When programming becomes boring, there is usually a construct that automates the writing. Computers are very good at performing repetitive tasks. For this purpose we use **loops**.

The `while` loop

A `while` -loop executes repeatedly a set of statements as long as a boolean `condition` is `True`

```
while condition:
    <statement 1>
    <statement 2>
    ...

<first statement after the loop>
```

Note that all statements to be executed within the loop must be indented by the same amount! The loop ends when an unindented statement is encountered.

In Python, indentations are very important. For instance, when writing a `while` loop:

In [54]:
```
counter = 0
while counter <= 10:
    counter = counter + 1

print(counter)
```

```
11
```

Let us take a look at what happens when we forget to indent correctly:

```
counter = 0
while counter <= 10:
counter = counter + 1
print(counter)
```

```
  File "<ipython-input-86-d8461f52562c>", line 3
    counter = counter + 1
    ^
IndentationError: expected an indented block
```

Let us now use the `while`-loop to create the table above:

In [55]:
```
C = -20  # Initialise C
dC = 5  # Increment for C within the loop
while C <= 40:  # Loop heading with condition (C <= 40)
    F = (9 / 5) * C + 32  # 1st statement inside loop
    print(C, F)  # 2nd statement inside loop
    C = C + dC  # Increment C for the next iteration of the loop.
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

Exercise 1.11: Compute the number of digits in a positive integer a

Write a Python function `num_digits(a)` that uses a `while` loop to compute how many decimal digits we need to write a positive integer a. For instance, we need 2 digits to write 73, whereas we need 5 digits to write 12345.

In [56]:
```python
# Uncomment and complete the code below. Do not change the names of variables!

def num_digits(a):
    count = 0
    if a == 0:
        return 1

    while a != 0:
        a //= 10  # remove the last digit
        count += 1
    return count
```

In [57]:
```python
with pybryt.check(pybryt_reference(1, 11)):
    num_digits(999_999_999)
```

```
REFERENCE: exercise-1_11
SATISFIED: True
MESSAGES:
  - SUCCESS: You are incrementing the number of digits in each iteration. Well done!
  - SUCCESS: Wow! Your function computes the number of digits correctly.
```

In [58]:
```python
import numbers
import numpy as np

res = num_digits(123_456_789)
assert isinstance(res, numbers.Real)
assert np.isclose(res, 9)

### BEGIN HIDDEN TESTS
assert callable(num_digits)

assert np.isclose(num_digits(0), 1)
assert np.isclose(num_digits(5), 1)
assert np.isclose(num_digits(10), 2)
assert np.isclose(num_digits(99), 2)
assert np.isclose(num_digits(100), 3)
### END HIDDEN TESTS
```

Exercise 1.12: Write an approximate Fahrenheit-Celsius conversion table

Instead of using an exact formula for converting Fahrenheit (F) to Celsius (C) degrees:

$$C = \frac{5}{9}(F - 32),$$

many people use an approximate formula for quicker conversion:

$$C \approx \hat{C} = \frac{F - 30}{2}.$$

Write a Python function `conversion_table` using a `while` loop that prints the conversion table consisting of three columns: F, C, and the approximate value $\hat{C}$, for $F = 0, 10, 20, \ldots, 100$. Besides, using the same while loop, count the number of rows printed and return the count. Ensure that all numbers in the conversion table are printed with two decimal places.

In [59]:
```
# Uncomment and complete the code below. Do not change the names of variables.

def conversion_table():
    count = 0
    F = 0
    dF = 10
    print("Fahrenheit  Celsius  Approximate")

    while F <= 100:
        C = 5/9 * (F-32)
        approx_C = (F-30) / 2
        print(f"{F:.2f}    {C:.2f}    {approx_C:.2f}")
        count += 1
        F += dF

    return count

rows = conversion_table()
print("Total rows:", rows)
```

```
Fahrenheit  Celsius  Approximate
0.00    -17.78    -15.00
10.00    -12.22    -10.00
20.00    -6.67    -5.00
30.00    -1.11    0.00
40.00    4.44    5.00
50.00    10.00    10.00
60.00    15.56    15.00
70.00    21.11    20.00
80.00    26.67    25.00
90.00    32.22    30.00
100.00    37.78    35.00
Total rows: 11
```

In [60]:
```
with pybryt.check(pybryt_reference(1, 12)):
    conversion_table()
```

```
Fahrenheit  Celsius  Approximate
0.00    -17.78    -15.00
10.00    -12.22    -10.00
20.00    -6.67    -5.00
30.00    -1.11    0.00
40.00    4.44    5.00
50.00    10.00    10.00
60.00    15.56    15.00
70.00    21.11    20.00
80.00    26.67    25.00
90.00    32.22    30.00
100.00    37.78    35.00
REFERENCE: exercise-1_12
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! In each iteration, you are increasing F by 10.
  - SUCCESS: Your computation of C is correct. Well done!
  - SUCCESS: Your computation of approximate C is correct.
  - SUCCESS: You are incrementing the number of rows in each iteration. Great!
  - SUCCESS: Wow! Your function returns the correct number of rows.
```

In [61]:
```
import numbers
import numpy as np
```

```
res = conversion_table()
assert isinstance(res, numbers.Real)
assert np.isclose(res, 11)

### BEGIN HIDDEN TESTS
assert callable(conversion_table)
### END HIDDEN TESTS
```

```
Fahrenheit  Celsius  Approximate
0.00    -17.78    -15.00
10.00    -12.22    -10.00
20.00    -6.67    -5.00
30.00    -1.11    0.00
40.00    4.44    5.00
50.00    10.00    10.00
60.00    15.56    15.00
70.00    21.11    20.00
80.00    26.67    25.00
90.00    32.22    30.00
100.00    37.78    35.00
```

Lists

So far, one variable has referred to one number (or string). Sometimes, however, we naturally have a collection of numbers, e.g. degrees −20, −15, −10, −5, 0, ..., 40. One way to store these values in a computer program would be to have one variable per value:

In [62]:
```
C1 = -20
C2 = -15
C3 = -10
...
C13 = 40
```

This is clearly a terrible solution, particularly if we have lots of values. A better way of doing this is to collect values together in a list:

In [63]:
```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Now, there is just one variable, `C`, holding all the values. We access elements in a list via an index. List indices are always *zero-indexed*, i.e. they are numbered as 0, 1, 2, and so forth up to the number of elements minus one:

In [64]:
```
mylist = [4, 6, -3.5]
print("First element:", mylist[0])
print("Second element:", mylist[1])
print("Third element:", mylist[2])
```

```
First element: 4
Second element: 6
Third element: -3.5
```

Here are a few examples of operations that you can perform on lists:

In [65]:
```
C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]
C.append(35)  # add new element 35 at the end
print(C)
```

```
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

In [66]:
```
C = C + [40, 45]  # And another list to the end of C
print(C)
```

```
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

In [67]:
```
C.insert(0, -15)  # Insert -15 as index 0
print(C)
```

```
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

In [68]:
```
del C[2]  # delete 3rd element
print(C)
```

```
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

In [69]:
```
del C[2]  # delete what is now 3rd element
print(C)
```

```
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

In [70]:
```
print(len(C))  # length of list
```

```
11
```

In [71]:
```
print(C.index(10))  # Find the index of the element with the value 10
```

```
3
```

In [72]:
```
print(10 in C)  # True only if the value 10 is stored in the list
```

```
True
```

In [73]:
```
print(C[-1])  # The last value in the list
```

```
45
```

In [74]:
```
print(C[-2])  # The second last value in the list
```

```
40
```

We can also extract sublists using `:`:

In [75]:
```
print(C[5:])  # From index 5 to the end of the list
```

```
[20, 25, 30, 35, 40, 45]
```

In [76]:
```
print(C[5:7])  # From index 5 up to, but NOT including index 7
```

```
[20, 25]
```

In [77]:
```
print(C[7:-1])  # From index 7 up to the second last element
```

```
[30, 35, 40]
```

In [78]:
```
print(C[:])  # [:] specifies the whole list.
```

```
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

We can also *unpack* the elements of a list into separate variables:

In [79]:
```
somelist = ["Curly", "Larry", "Moe"]
stooge1, stooge2, stooge3 = somelist
print(f"stooge1: {stooge1}, stooge2: {stooge2}, stooge3: {stooge3}")
```

```
stooge1: Curly, stooge2: Larry, stooge3: Moe
```

Exercise 1.13: Store odd numbers in a list

Write a Python function `odd_numbers(n)` that uses a `while`-loop, and generates and returns a list of all odd numbers from 1 to n. (Make sure that if n is an even number, the largest generated odd number is $n-1$.)

In [80]:
```
# Uncomment and complete code. Do not change the variable names.

def odd_numbers(n):
    lis = []
    number = 1
    while number <= n:
        lis.append(number)
        number += 2
    return lis
```

In [81]:
```
with pybryt.check(pybryt_reference(1, 13)):
    odd_numbers(50)
```

```
REFERENCE: exercise-1_13
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! In each iteration, you are appending an odd number to the list.
  - SUCCESS: In each iteration, you are increasing the number by 2.
  - SUCCESS: Before the loop, you begin with an empty list. Well done!
  - SUCCESS: Before the loop, you start with 1.
  - SUCCESS: Your function returns the correct solution.
```

In [82]:
```
import numbers
import numpy as np

res = odd_numbers(10)
assert isinstance(res, list)
assert len(res) == 5
assert np.allclose(res, [1, 3, 5, 7, 9])

### BEGIN HIDDEN TESTS
assert callable(odd_numbers)
```

```
assert all(i % 2 for i in odd_numbers(200))
### END HIDDEN TESTS
```

`for`-loops

We can visit each element in a list and process it with some statements using a `for`-loop, for example:

In [83]:
```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print("list element:", C)
print(f"The list has {len(degrees)} elements.")
```

```
list element: 0
list element: 10
list element: 20
list element: 40
list element: 100
The list has 5 elements.
```

Notice again how the statements to be executed within the loop must be indented! Let us now revisit the conversion table example using the `for` loop:

In [84]:
```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9 / 5) * C + 32
    print(C, F)
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

We can easily beautify the table using the *printf syntax* we encountered previously:

In [85]:
```
for C in Cdegrees:
    F = (9.0 / 5) * C + 32
    print(f"{C:5d} {F:5.1f}")
```

```
 -20  -4.0
 -15   5.0
 -10  14.0
  -5  23.0
   0  32.0
   5  41.0
  10  50.0
  15  59.0
  20  68.0
  25  77.0
  30  86.0
  35  95.0
  40 104.0
```

It is also possible to rewrite the `for` loop as a `while` loop, i.e.

```
for element in somelist:
    # process element
```

can always be transformed to a `while` loop

```
index = 0
while index < len(somelist):
    element = somelist[index]
    # process element
    index += 1
```

Let us see how a previous example would look like if we used `while` instead of `for` loop:

In [86]:
```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0 / 5) * C + 32
    print(f"{C:5d} {F:5.1f}")
    index += 1
```

```
 -20  -4.0
 -15   5.0
 -10  14.0
  -5  23.0
   0  32.0
   5  41.0
  10  50.0
  15  59.0
  20  68.0
  25  77.0
  30  86.0
  35  95.0
  40 104.0
```

Rather than just printing out the Fahrenheit values, let us also store these computed values in a list of their own:

In [87]:
```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
Fdegrees = []  # start with empty list
for C in Cdegrees:
    F = (9 / 5) * C + 32
    Fdegrees.append(F)  # add new element to Fdegrees
print(Fdegrees)
```

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0, 104.0]
```

In Python, `for` loops usually loop over list values (elements), i.e.

```
for element in somelist:
    # process variable element
```

However, we can also loop over list indices:

```
for i in range(0, len(somelist), 1):
    element = somelist[i]
    # process element or somelist[i] directly
```

The statement `range(start, stop, increment)` generates a list of integers *start*, *start+increment*, *start+2*\increment*, and so on up to, but not including, *stop*. We can also write `range(stop)` as an abbreviation for `range(0, stop, 1)`:

In [88]:
```
for i in range(3):  # same as range(0, 3, 1)
    print(i)
```

```
0
1
2
```

In [89]:
```
for i in range(2, 8, 3):
    print(i)
```

```
2
5
```

List comprehensions

Consider this example where we compute two lists in a `for` loop:

In [90]:
```
n = 16

# empty lists
Cdegrees = []
Fdegrees = []

for i in range(n):
    Cdegrees.append(-5 + i * 0.5)
```

```
    Fdegrees.append((9 / 5) * Cdegrees[i] + 32)

print("Cdegrees = ", Cdegrees)
print("Fdegrees = ", Fdegrees)
```

```
Cdegrees =  [-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5]
Fdegrees =  [23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1, 32.0, 32.9, 33.8, 34.7, 35.6, 36.5]
```

As constructing lists is a very common task, the above way of doing it can become very tedious both to write and read. Therefore, Python has a compact construct, called *list comprehension* for generating lists from a `for` loop:

In [91]:
```
n = 16
Cdegrees = [-5 + i * 0.5 for i in range(n)]
Fdegrees = [(9 / 5) * C + 32 for C in Cdegrees]
print("Cdegrees = ", Cdegrees)
print("Fdegrees = ", Fdegrees)
```

```
Cdegrees =  [-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5]
Fdegrees =  [23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1, 32.0, 32.9, 33.8, 34.7, 35.6, 36.5]
```

The most general form of a list comprehension is:

```
somelist = [expression for element in somelist if condition]
```

Here the `condition` can be used to pick out elements which satisfy a specific property.

Exercise 1.14: Create a list of even numbers ranging from 0 to n using a `for` loop.

Write a function `even_numbers(n)` using a `for` loop, which generates and returns a list of all even numbers from 0 to n. For instance, `even_numbers(10)` should return list `[0, 2, 4, 6, 8]`. In this exercise, do not use list comprehensions.

In [92]:
```
# Uncomment and complete code. Do not change the variable names.

def even_numbers(n):
    lis = []
    i = 0
    for i in range(n):
```

```
        if i % 2 == 0:
            lis.append(i)
            i += 2
    return lis
```

In [93]:
```
with pybryt.check(pybryt_reference(1, 14)):
    even_numbers(11)
```

```
REFERENCE: exercise-1_14
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! In each iteration, you are appending an even number to the list.
  - SUCCESS: You are iterating over the correct list.
  - SUCCESS: Before the loop, you begin with an empty list. Well done!
  - SUCCESS: Your function returns the correct solution.
```

In [94]:
```
import numbers
import numpy as np

res = even_numbers(10)
assert isinstance(res, list)
assert len(res) == 5
assert np.allclose(res, [0, 2, 4, 6, 8])

### BEGIN HIDDEN TESTS
assert callable(even_numbers)
assert all(i % 2 == 0 for i in even_numbers(200))
### END HIDDEN TESTS
```

Exercise 1.15: Implement the sum function

The built-in Python function sum takes a list as an argument and computes the sum of the elements in the list:

```
>> sum([1, 3, 5, -5])
4
```

Implement your own version of the sum function and name it `my_sum`.

In [95]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
def my_sum(lst):
    total = 0
    for element in lst:
        total += element
    return total

print(my_sum([1, 3, 5, -5]))
```

```
4
```

In [96]:
```
with pybryt.check(pybryt_reference(1, 15)):
    my_sum([2.1, 98, -451, 273, 1111, 23.98])
```

```
REFERENCE: exercise-1_15
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! In each iteration, you are adding a list element to the sum variable.
  - SUCCESS: You are iterating over the list elements instead over indices.
  - SUCCESS: Before the loop, you set the sum to be zero. Well done!
  - SUCCESS: Your function returns the correct solution.
```

In [97]:
```
import numbers
import numpy as np

res = my_sum([-1, 1, 2, 3, 0.1])
assert isinstance(res, numbers.Real)
assert np.isclose(res, 5.1)

### BEGIN HIDDEN TESTS
assert callable(my_sum)
assert np.isclose(my_sum([]), 0)
assert np.isclose(my_sum(range(1, 101)), (100 * 101) / 2)
### END HIDDEN TESTS
```

Exercise 1.16: Position of the ball in a vertical motion.

Write a function `distance` that returns a `list` of y values calculated using the formula:

$$y(t) = v_0t - {1\over2} gt^2,$$

for `n` number of values t ranging from `t_start` to `t_end`. Specify the keyword arguments `v0=6.0` and `g=9.81`.

In [98]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
def distance(t_start, t_end, n, v0=6.0, g=9.81):
    if n == 1:
        t_values = [t_start]
    else:
        dt = (t_end - t_start) / (n - 1)
        t_values = [t_start + i*dt for i in range(n)]

    distances = []
    for t in t_values:
        s = v0 * t - 0.5 * g * t**2
        distances.append(s)

    return distances

print(distance(0, 10, 5))
```

```
[0.0, -15.65625, -92.625, -230.90625, -430.5]
```

In [99]:
```
with pybryt.check(pybryt_reference(1, 16)):
    distance(0, 10, 5)
```

```
REFERENCE: exercise-1_16
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You are iterating over the correct values of t.
  - SUCCESS: You are appending the positions to a list. Well done!
  - SUCCESS: Before the loop, you start with an empty list. Amazing!
  - SUCCESS: Your time step value is correct.
  - SUCCESS: Your function returns the correct solution.
```

In [100…
```
import numbers
import numpy as np

res = distance(0, 10, 10)
assert isinstance(res, list)
assert all([isinstance(i, numbers.Real) for i in res])
```

```
assert np.isclose(res[0], 0)

### BEGIN HIDDEN TESTS
assert callable(distance)
assert np.allclose(distance(0, 10, 5), [0.0, -15.65625, -92.625, -230.90625, -430.5])
### END HIDDEN TESTS
```

Exercise 1.17: Cumulative sum

Write a function that returns the cumulative sum of the numbers in a list. The function should return a list, whose `i`-th element is the sum of the input list up to and including the `i`-th element.

For example, for the list `[1, 4, 2, 5, 3]` should return `[1, 5, 7, 12, 15]`.

In [101…

```
# Uncomment and complete this code - keep the names the same for testing purposes.

def my_cumsum(x):
    cum_sum = []
    total = 0

    for i in x:
        total += i
        cum_sum.append(total)

    return cum_sum

print(my_cumsum([55, 111, -33, 65]))
```

```
[55, 166, 133, 198]
```

In [102…

```
with pybryt.check(pybryt_reference(1, 17)):
    my_cumsum([55, 111, -33, 65])
```

```
REFERENCE: exercise-1_17
SATISFIED: True
MESSAGES:
  - SUCCESS: You are appending the correct cumulative sum to a list. Well done!
  - SUCCESS: Before the loop, you start with an empty list. Amazing!
  - SUCCESS: Wow! Your function returns the correct solution.
```

In [103…
```
import numbers
import numpy as np

res = my_cumsum(range(10))
assert isinstance(res, list)
assert all([isinstance(i, numbers.Real) for i in res])
assert np.isclose(res[0], 0)
assert np.isclose(res[-1], 45)

### BEGIN HIDDEN TESTS
assert callable(my_cumsum)
assert np.allclose(my_cumsum(range(100)), np.cumsum(range(100)))
### END HIDDEN TESTS
```

Exercise 1.18: Bouncing ball

A rubber ball is dropped from a height `h_0`. After each bounce, the height it rebounds to decreases by 10%, i.e. after one bounce, it reaches `0.9*h_0`, after two bounces it reaches `0.9*0.9*h_0`, etc. Write a Python function `compute_heights` that returns a list of the maximum heights of the ball after each bounce (including after 0 bounces, i.e. its initial height), until either the ball has bounced `n` times *or* its maximum height falls below `h1`. The function should take `h_0`, `h_1` and `n` as keyword arguments, with default values of `1.0`, `0.3` and `10`, respectively.

HINT: If `n` bounces of the ball are not reached, the last value of height in the list should be the first height at which the ball fell below `h1`. More precisely, the last value is less than `h1`.

In [104…
```
# Uncomment and complete this code - keep the names the same for testing purposes.
def compute_heights(h_0=1.0, h_1=0.3, n=10):
    heights = []
    current_height = h_0
    count = 0

    while count <= n:
        if count == 0:
            heights.append(current_height)
        else:
            current_height *= 0.9
```

```
            heights.append(current_height)
            if current_height < h_1:
                break
        count += 1

    return heights

print(compute_heights())
```

```
[1.0, 0.9, 0.81, 0.7290000000000001, 0.6561000000000001, 0.5904900000000002, 0.5314410000000002, 0.47829690000000014, 0.4304672
1000000016, 0.38742048900000015, 0.34867844010000015]
```

In [105…

```
with pybryt.check(pybryt_reference(1, 18)):
    compute_heights(h_0=1.0, h_1=0.3, n=10)
```

```
REFERENCE: exercise-1_18
SATISFIED: True
MESSAGES:
  - SUCCESS: You are appending the correct height to a list. Well done!
  - SUCCESS: You are counting the bounces correctly. Well done!
  - SUCCESS: Before the loop, you start with an empty list. Amazing!
  - SUCCESS: Before the loop, you set the ounter to zero. Great!
  - SUCCESS: Wow! Your function returns the correct solution.
```

In [106…

```
import numbers
import numpy as np

res = compute_heights(h_0=1.0, h_1=0, n=10)
assert isinstance(res, list)
assert all([isinstance(i, numbers.Real) for i in res])
assert np.isclose(res[0], 1)
assert np.isclose(res[-1], 0.9**10)

### BEGIN HIDDEN TESTS
assert callable(compute_heights)
assert np.allclose(
    compute_heights(h_0=1.0, h_1=0.5, n=10),
    [
        1.0,
        0.9,
        0.81,
```

```
        0.7290000000000001,
        0.6561000000000001,
        0.5904900000000002,
        0.5314410000000002,
        0.47829690000000014,
    ],
)
### END HIDDEN TESTS
```

Exercise 1.19: Calculate π

A formula for π is given by the *Gregory-Leibniz series*:

$$\pi = 4 \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + ... \right)$$

Note that the denominators of the terms in this series are the positive odd numbers. Write a Python function `calculate_pi(n)` following the guidelines below; each of the first three steps can be completed using a single list comprehension.

Step 1:

Produce a list of the first `n` odd numbers, for `n=100`.

Step 2:

Make a list of the signs of each term, i.e. `[1, -1, 1, -1, ...]`. Hint: think about the value of $(-1)^i$ for integer i.

Step 3:

Using the results of steps 1 and 2, make a list of the first `n` terms in the above series.

Step 4:

Use your `my_sum` function to sum this series, multiply by 4, and return the result.

In [107…

```
def my_sum(lst):
    total = 0
    for x in lst:
```

```
        total += x
    return total

def calculate_pi(n):
    # Step 1:
    odd_numbers = [2*i + 1 for i in range(n)]

    # Step 2:
    signs = [(-1)**i for i in range(n)]

    # Step 3:
    terms = [signs[i] / odd_numbers[i] for i in range(n)]

    # Step 4:
    pi = 4 * my_sum(terms)
    return pi

print(calculate_pi(100))
```

```
3.1315929035585537
```

In [108…

```
with pybryt.check(pybryt_reference(1, 19)):
    calculate_pi(100)
```

```
REFERENCE: exercise-1_19
SATISFIED: True
MESSAGES:
  - SUCCESS: You generated odd integers correctly. Amazing!
  - SUCCESS: You generated the signs list correctly. Well done!
  - SUCCESS: Great! You computed series terms correctly.
  - SUCCESS: Wow! Your function returns the correct solution.
```

In [109…

```
import numbers
import numpy as np

res = calculate_pi(1000)
assert isinstance(res, numbers.Real)
assert np.isclose(res, np.pi, rtol=1e-3)

### BEGIN HIDDEN TESTS
assert callable(calculate_pi)
```

```
assert np.isclose(calculate_pi(1), 4)
assert np.isclose(calculate_pi(0), 0)
### END HIDDEN TESTS
```

In [ ]: