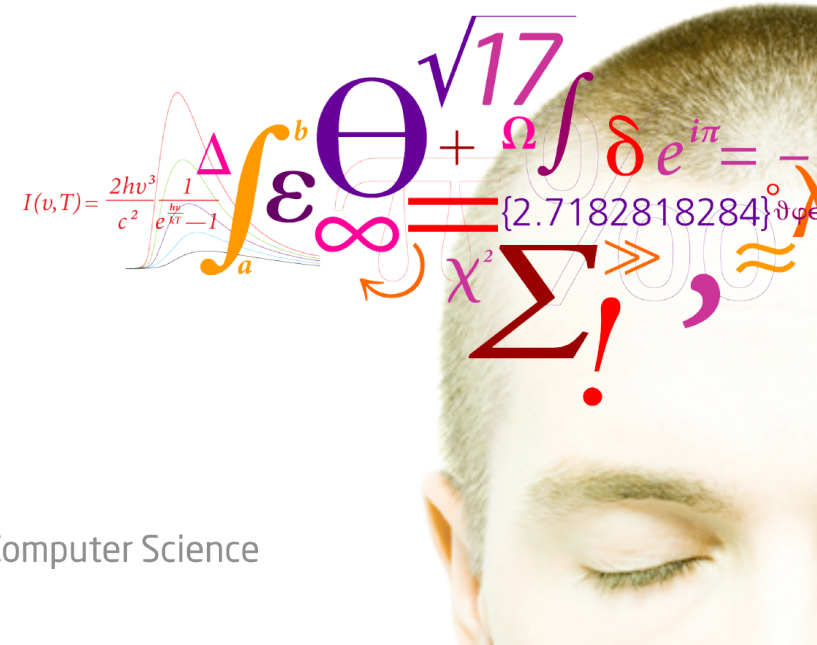
Introduction to programming and data processing

with Python

Exercises for DTU Course 02631–34, 02691–94.

Mikkel N. Schmidt, Kristoffer Albers, Rasmus Røge,
Vedrana Andersen, Martin S. Andersen.

2018



DTU Compute
Department of Applied Mathematics and Computer Science

Contents

Introduction to programming

Aims and objectives

After working through this exercise you should be able to:

- Run the Python desktop environment (Spyder).
- Use the desktop envionment including
  - The console window where you can type commands.
  - The variable explorer window where you can inspect the variables in your workspace.
  - The editor window where you can edit scripts and functions.
  - The file explorer window where you can find files with code saved on the computer.
- Use the desktop environment to get help on functions.
- Type and execute commands in the command prompt (console window).
- Create variables and assign values of different types including
  - Decimal numbers
  - Whole numbers
  - Text strings
  - Logical values (True/False).
- Use the basic arithmetic operators to add, subtract, multiply, divide, and exponentiate variables.
- Use simple functions such as cos(), sin(), tan(), exp(), and log().
- Write a script containing multiple statements and run it in the desktop environment.
- Save scripts in a file and open and run scripts saved in files.

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 1–2.

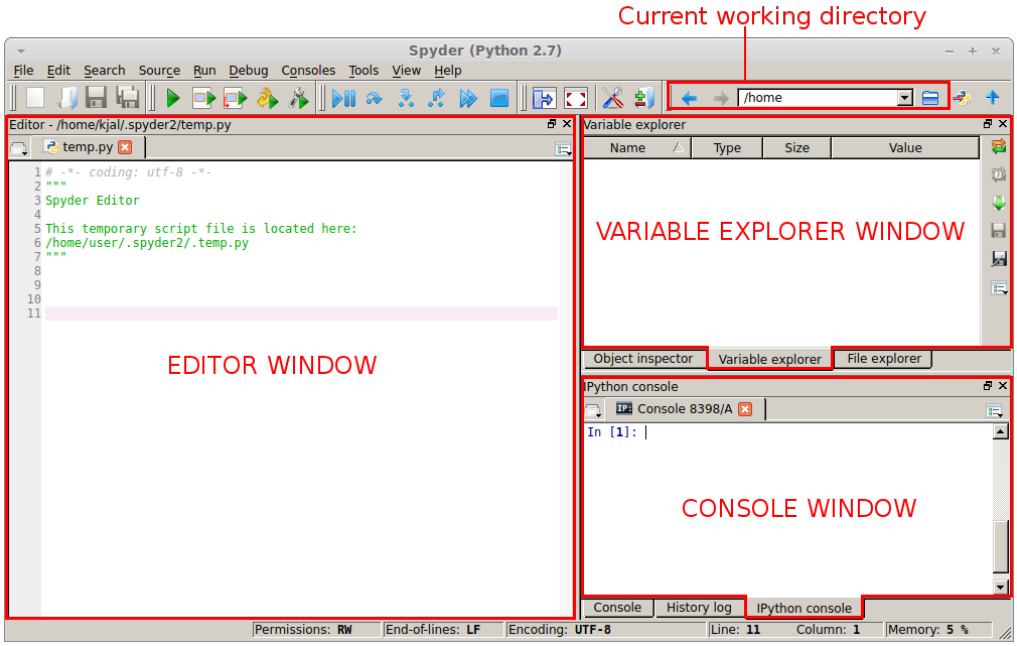
Video: Using the desktop environment

Installation

Python can be downloaded from `http://continuum.io/downloads` Choose the newest version.

The integrated development environment interface

When you start Spyder, you will be met by an interface, similar to the following.



The interface is split into several windows.

- The *console window* is used to prompt commands, that you want Python to execute. Make sure the 'iPython console' tab is selected to show the correct window.
- The *variable explorer window* shows a list of all created variables. The list is initially empty, as no variables have yet been created. Make sure the 'Variable explorer' tab is selected to show the correct window.
- The *file explorer window* shows a list of files present in the *current working directory*. You must select the 'File explorer' tab to show the file explorer window, next to the 'Variable explorer' tab.

Exercise 1A Getting to know the interface

1. Click with the mouse in the console window. Type `a = 3` and press enter. You have now prompted Python to create a variable called `a`, and given it the initial value 3. Notice that the variable `a` is now shown in the variable explorer window.
2. Type `a+2` in the console window and press enter. Python now shows the result of the arithmetic operation of adding 2 to the value of `a`. Notice in the variable explorer window that the value of `a` is unchanged.
3. Type `a = a + 2` and press enter. This instructs Python to perform the arithmetic operation and write the result to the variable `a`.
   - What it the value of `a` now? What has happened with the old value of `a`?
4. Use the console window to perform the following operations:
   - Create a new variable `x` with the value 4.
   - Create a new variable `y` with the value 7.
   - Create a new variable `z` and set its value to the sum of `x` and `y`.
   - Inspect the variable explorer window to ensure that the value of `z` is in fact 11.
5. If you type in the following sequence of commands,

```
a = 7
b = a
a = 9
```

what will the final value of the variable `b` be? Check your answer by typing it into the console window. If it is not clear to you why `b` still has the value 7 even though `a` has been assigned a new value 9, you need to read up on how *variable assignment* works.

1A

Representing and displaying numbers

When we do computations with numbers in Python, we usually work with the socalled *double-precision floating-point* format. Numbers are represented with a finite number of significant digits, corresponding to approximately 16 decimals. This means that numbers that can not be represented by a finite number of binary digits will automatically be rounded. Usually this is no problem, because 16 significant digits is most often more than enough, but sometimes the finite precision can be a bit confusing: For example, try typing the following into Python and see what the result is

```
1.1 + 2.2 - 3.3
```

> **Hint**
>
> Python always displays decimal numbers with at least one decimal, even if the decimal is zero. For example, the number 12 will be displayed as `12.0`.

Exercise 1B Simple arithmetic operations

The arithmetic operators `+`, `-`, `*`, `/` are used to add, subtract, multiply and divide.

> **Hint**
>
> In some older versions of Python, division of two integer numbers will always result in an integer number. For example the expression 9/2 will give the result 4. To avoid this behaviour, just one of the operands must be a real number. For instance 9.0/2 will give the result 4.5 as wanted. Write the two expressions 9/2 and 9.0/2 in the console window to determine the behaviour of your installed Python interpreter.

1. What are the results of the following expressions? Think about it first, then type the expressions in the console window to verify.

   ```
   9 / 2 + 2
   ```

   ```
   6 + 4 * 4
   ```

   ```
   4 / 6 + 2
   ```

   ```
   2 - 3 * 2 - 4
   ```

   ```
   (7 - 5) * (7 + 5)
   ```

   ```
   (3 * (4 - 2 * (3) + 4))
   ```

2. The exponentiation operator in Python is `**`. What do you expect the results of the following expressions to be? Verify, by writing the expressions in the console window.

   ```
   2**3
   ```

   ```
   3**2 - 4**2
   ```

   ```
   2**2**3
   ```

   ```
   2**(2**3)
   ```

3. Verify in Python that the following expressions are correct.

   $3 \cdot (2 + 3^2) = 33$

   $4^2 + 3^3 + 2^4 + 1 = 60$

$$\frac{1}{\frac{1}{2}+\frac{1}{3}+\frac{1}{6}} = 1$$

1B

Exercise 1C Help on functions

> **Hint**
>
> Much of the functionality in Python must be explicitly imported into the interpreter before it can be used. Write `import math` in the console and press enter. The math module is now imported. It contains many mathematical functions and variables that you can now use.

The `help` command is useful for learning more about functions and how they are used. For example, writing `help(math.sin)` in the console window shows a helpful description of the sinus function.

1. Use the help to confirm that the function calculates and returns the sine of an angle in *radians* and not in degrees.

   Next, test this in the console window by writing both of the following lines and inspecting the results.

   ```
   math.sin(0.5*math.pi)
   math.sin(90)
   ```

2. Use the help to examine what the functions `round`, `math.floor`, `math.ceil`, and `math.trunc` do.

   Next, verify your findings by answering the following questions:

   - Is `math.floor(2.5)` the same as `math.trunc(2.5)`?
   - Is `math.ceil(-3.6)` the same as `math.trunc(-3.2)`?
   - Is `round(3.4)` the same as `math.ceil(3.4)`?
   - Is `round(-3.4)` the same as `math.ceil(-3.4)`?

3. The `round` function can also take a second argument, specifying how many decimal places to round off to. Try using this to round of the number 1234.56789 to the following values:

   - 1234.568
   - 1234.6
   - 1235
   - 1200

4. The function `math.log` can be used to compute the logarithm of a number. There exists different logarithm functions with different *base*, such as the base-10 logarithm and the natural logarithm where the base is Euler's number, e. Use the help to find out what the base is in the `math.log` function.

1C

Exercise 1D Mathematical functions

1. Write the following lines in the console window and observe what they do:

   ```
   math.pi
   math.sqrt(25)
   ```

   The radius r of a circle can be found when the area A is known as:

   $$r = \sqrt{\frac{A}{\pi}} \tag{1.1}$$

How can you calculate the radius of a circle with area $A = 30$?

Solution

```
math.sqrt(30/math.pi)
```

2. Type `a = 51` and press enter. Now the variable `a` contains the value `51`. Type `b = math.log(a)` to get Python to compute the logarithm of 51. What do the variables `a` and `b` contain now?

   Next, type `a = "fiftyone"`. Again, type `b = math.log(a)`. This time you will get an error message—why? Confirm that since the command failed, the variable `b` whas not changed.

1D

Exercise 1E Scripts

A script is simply a textfile, where each line contains a command. When the script is run, the commands are executed in sequence. This is essentially the same as prompting the commands one at a time in the console window.

There are however some advantages of using scripts. They give you a more clear overview of your code, lets you save the sequence of commands for future use and makes it easy to share your code with other people.

A script must have the extention `.py` (for example `MyScript.py`) and is therefore sometimes referred to simply as a py-file. Scripts can be created with the built-in editor.

- Create a new empty file in the editor window.
- In the empty file, write the three commands you used to solve exercise 1A part 4.
- Save your script (for instance as myFirstScript.py). Notice that the script is now listed in the file explorer window.
- Execute the commands in your script. This can be done by pressing F5 from within the editor.
- Save your script under a different name, and observe that both script-files are now present in the file explorer window.

1E

Assignment 1F Hello CodeJudge!

Create a script in which you create a variable named `WhatISay` and assign to it the string

```
Hello CodeJudge!
```

Be careful to name your variable exactly as specified, and assign the string with correct case and including the exclamation point.

Save your script in a file.

> **Hint**
>
> It is important that your script does not print out anything to the screen or does anything else that is not specified.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(WhatISay)`	`Hello CodeJudge!`

Hand in on CodeJudge

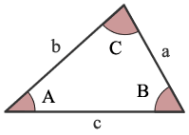
Hand in your solution on CodeJudge.

1F

Assignment 1G The cosine rule

The cosine rule can be used to compute the length of a side in a triangle (a), when you know the other two sides (b, and c) and the opposing angle (A):

$$a = \sqrt{b^2 + c^2 - 2bc\cos(A)} \tag{1.2}$$



Problem definition

Create a script that carries out the following steps:

1. Create a variable `b` and set it to 12.
2. Create a variable `c` and set it to 10.
3. Create a variable `A` and set it to $0.25 \cdot \pi$.
4. Compute the length of the side a using the cosine rule.
   - Save the solution in a variable `a`.

Check that your script produces the correct results, namely $a = 8.6194$ (here show with four decimals).

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(a)`	`8.619418339727373`

Hand in on CodeJudge

Your script must be handed in on CodeJudge.

1G

Assignment 1H The quadratic formula

A *quadratic equation* has the following form

$$ax^2 + bx + c = 0. \tag{1.3}$$

Solutions to the equation can be found using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{1.4}$$

Problem definition

Create a script that carries out the following steps:

1. Create a variable `a` and set it to 2.
2. Create a variable `b` and set it to -5.
3. Create a variable `c` and set it to 2.
4. Compute the two solutions using the quadratic formula.
   - Save the smallest solution in a variable `x1`.
   - Save the largest solution in a variable `x2`.

Check that your script produces the correct results, namely 0.5 and 2.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(a)`	`2`

Hand in on CodeJudge

Your script must be handed in on CodeJudge.

1H

Functions, vectors, and testing

Aims and objectives

After working through this exercise you should be able to:

- Create your own user-defined functions that accept a single input and produces a single output.
  - Execute the functions both from the console and from within a script.
- Describe the difference between a script and a function.
- Describe how the *scope* of variables works. In particular you should be able to explain:
  - Which variables can be accessed from a script and which can be accessed from within a function?
  - What is the relation between a variable that is passed as an input argument to a function, and the corresponding variable inside the function?
- Initialize a vector and do basic vector operations:
  - Indexing, including logical indexing.
  - Elementwise operations such as division and multiplication.
  - Vector operations such as addition, subtraction, dot products, and outer products.
  - Operations that work on vectors, such as computing the sum, mean, maximum, minimum, and product of the elements in the vector.
  - Determine the length of a vector.
- Understand simple error messages that occur for example in the following situations:
  - Calling functions with a wrong number of arguments.
  - When trying to do vector operations that are not possible.

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 3.

Video: Working with functions
Video: Indexing vectors
Video: Assert statements and unit tests

Functions and scope

A script and a function are quite similar: They both contain multiple lines of code that can be executed. The main difference lies in which variables a script and a function have access to. A script has access to all variables in the workspace (these are often called *global variables*) whereas a function primarily has access to variables that are defined within the function (these are often called *local variables*). The *scope* of a variable is the part of your program in which the variable is accesible.

While it is possible to access global variables from within a function, it is often not a good idea. The structure of your program is more clear, if functions only work with variables that are defined within the function itself. Variables that you need from outside should be passed as input arguments, and variables containing the results of the computations in the function should be passed as return-values.

You can think of a variable as a *name* that refers to a *value*. In two different functions it is allowed to use the same name to refer to different values: For example, in one function you might have a variable called `data` which is equal to the number `13`, and in another function you might have a variable which is also called `data` which is equal to the string `datafile.txt`. This is no problem, because the two variables only exist inside the two functions, and can not be directly accessed from outside: They are only accesible inside the function scope.

Creating your own functions in Python is easy. For example, the following defines a function which evaluates the polynomial $5x^2 - 7x + 3$.

```
def evaluate_polynomial(x):
    a = 5
    b = -7
    c = 3
    return a*x**2 + b*x + c
```

To use the function you must save it in a file and run it. Now, the function `evaluate_polynomial` can be used in the console window, in a script, or in another function.

> **Hint**
>
> In general, to use a function `myfunction` which is stored in a file `myfunction.py`, you must import the function into Python, either by running the file or by writing `from myfile import myfunction`.

We can for example type the following in the console window:

```
>> t = 3
>> x = evaluate_polynomial(t)
>> x
27
```

Note that in the console window, the global variable `t` is has the value `3` and the global variable `x` gets the value `27`. Inside the function, the local input variable is *also* called `x`, but since this is inside the function it is a different `x`. The local `x` gets its value from the argument to the function which is `t` equal to 3. Thus, inside `evaluate_polynomial`, `x` is equal to 3.

Printing versus returning values

It is important to distinguish between *printing* a value and *returning* a value in a function. Compare the following code to the previous implementation of the `evaluate_polynomial` function:

```
def evaluate_polynomial(x):
    a = 5
    b = -7
    c = 3
    print(a*x**2 + b*x + c)
```

In this second version, the value of the polynomial is not returned from the function, but is printed to the screen. A common mistake is to print out a value which should have been returned from a function. Printing something to the should screen only be done to display information to the user of the computer program. If you compute something that you want to be able to use on other parts of a computer program, it should be returned rather than printed.

Returning multiple values

It is also possible to return more than one value from a function. You can do this by returning a *tuple*, i.e. multiple values surrounded by parentheses and separated by commas.

```
def painting():
    name = "Mona Lisa"
    value = 100000000
    return (name, value)
```

When you call the function, you can then access the two return values like this:

```
>>> artName, artVal = painting()
>>> artName
'Mona Lisa'
>>> artVal
100000000
```

Assignment 2A Taylor series

A *Taylor series* is a way to represent a complicated mathematical function as a an infinite sum of terms. Often, a Taylor series can be used to approximate a mathematical function by computing a small number of the terms and ignoring the remaining. For example, the first three terms of the Taylor series for the $\log(x)$ function are given by the following polynomial expression:

$$y = (x-1) - \tfrac{1}{2}(x-1)^2 + \tfrac{1}{3}(x-1)^3 \tag{2.1}$$

Problem definition

Create a function named `evaluateTaylor` that evaluates the Taylor series approximation in Equation (2.1) at an input x and returns the result.

Solution template

```
def evaluateTaylor(x):
  # Insert your code here
  # y = ... ?
  return y
```

Input	
`x`	Input x (real scalar value)
Output	
`y`	Result of Taylor series at x (real scalar values)

Example

Evaluating the Taylor series at $x = 1$ should give the result $y = 0$, since all three terms are zero in that case. Evaluating the series at $x = 2$ yields

$$y = (2-1) - \tfrac{1}{2}(2-1)^2 + \tfrac{1}{3}(2-1)^3 = 1 - \tfrac{1}{2} + \tfrac{1}{3} = \tfrac{5}{6} \approx 0.833. \tag{2.2}$$

You can use these examples to validate your code before you submit your solution to CodeJudge.

> **Hint**
>
> Note that in Python the indentation whitespace (spaces, tabs, and newlines) is what determines the code sections. Therefore wrongly indented code will often not run. The following two implementations of the function `multiply_by_three` will not work due to wrong indentation
>
> ```
> # Will NOT work
> def multiply_by_three(n):
>     m = 3*n
>   return m
>
> # Will NOT work
> def multiply_by_three(n):
> m = 3*n
> return m
> ```

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(evaluateTaylor(2))`	`0.8333333333333333`

Hand in on CodeJudge

In order to validate you can either upload a file containing your implementation of the function or directly copy the code into codejudge.

Discussion and futher analysis

- Run the function from the console with different input.
- What happens when you run the function with a vector as input?
- What happens when you try to run the function without input.

Working with vectors

The following is a short introduction to working with vectors in Python.

The NumPy module To work with vectors and matrices in Python, we will use a package for scientific computing called NumPy. It is included by default in most Python installations, but if you have a version of Python that does not include NumPy it can be downloaded and manually installed from www.scipy.org/Download.

The most common way to import the NumPy packages is

```
import numpy as np
```

This imports all the libraries and gives it the short name `np`. All functions in NumPy must then be preceeded with the `np` keyword. Alternatively you can write

```
from numpy import *
```

This means you can now access NumPy functions without using any keywords. While this might seem easier, it will make it difficult to tell which library a specific function belongs and this method is therefore not recommended. Further documentation on NumPy can be obtained from the NumPy reference guide available online: docs.scipy.org/doc/numpy/reference. There is also a simple but more extensive guide to NumPy available from www.engr.ucsb.edu/~shell/che210d/numpy.pdf. Note that all the functions mentioned here have additional functionality. For a complete description of the functionality and notation of a function visit the NumPy reference guide. For the rest of this section we assume that NumPy has been included by `import numpy as np`.

Creating vectors A vector is called an *array* in NumPy and can be created by converting a list of numbers (in square brackets) using the `np.array` function.

```
a = np.array([1, 2, 3])
```

Accessing elements The elements can be accessed using square brackets. The index of the first element of a vector is 0 and the index of the last element is N-1 where N is the length of the vector.

```
>> a[0]
1
```

You can also use this notation to change an element in a vector.

```
>> a = np.array([1, 2, 3])
>> a[0] = 7
>> a
array([7, 2, 3])
```

You can use the colon operator as an index in a vector to extract a sub-range.

```
>> a = np.array([3.4, 5.2, 7.4, 9.2, 5.3])
>> a[1:4]
array([ 5.2,  7.4,  9.2])
```

You can also access multiple elements in a vector by using a vector of indexes.

```
>> index = np.array([0, 2, 4]);
>> a[index]
array([ 3.4,  7.4,  5.3])
```

You can also use the colon operator or a vector of indices to change multiple elements in a vector.

```
>>> a = np.array([3.4, 5.2, 7.4, 9.2, 5.3]);
>>> index = np.array([0, 2, 4])
>>> a[index] = 0
>>> a
array([ 0. ,  5.2,  0. ,  9.2,  0. ])
>>> a[0:3] = np.array([10, 9, 8])
>>> a
array([ 10. ,   9. ,   8. ,   9.2,   0. ])
```

Vectors of identical numbers It is easy to create vectors with identical values, such as all zeros or ones.

```
>> a = np.zeros(4)
>> a
array([ 0.,  0.,  0.,  0.])
>> b = np.ones(4)
>> b
array([ 1.,  1.,  1.,  1.])
```

Number ranges A vector that contains a range of numbers can be constructed using the `arange` function.

```
>> a = np.arange(5, 10)
>> a
array([5, 6, 7, 8, 9])
```

Note that you specify the start and stop values, and that the range does not include the stop value.

Type `help(np.arange)` to learn more about this.

Concatenating vectors You can concatenate two vectors using the `concatenate` function.

```
>> a = np.arange(1,4)
>> b = np.arange(6,8)
>> c = np.concatenate((a,b))
>> c
array([1, 2, 3, 6, 7])
```

Random vectors To create a random vector you can use the function `np.random.rand`.

```
>>> np.random.rand(4)
array([ 0.45522641,  0.36673742,  0.11908726,  0.25607802])
```

This will generate random decimal numbers between 0 and 1 with a uniform distribution. There are also functions that can generate random integers, and random numbers with other probability distributions—you can use the help and documentation to learn more.

Element-wise vector operations You can use the operators `+`, `-`, `*`, `/`, and `**` to perform element-wise addition, subtraction, multiplication, division, and exponentiation of vectors.

```
>> a = np.array([1, 2, 3, 4, 5])
>> a*2
array([ 2,  4,  6,  8, 10])
>> 2+a
array([3, 4, 5, 6, 7])
>> a/3
array([ 0.33333333,  0.66666667,  1.        ,  1.33333333,  1.66666667])
```

This also applies for when both operands are vectors.

```
>> b = np.array([5, 4, 3, 2, 1])
>> a+b
array([6, 6, 6, 6, 6])
>> a-b
array([-4, -2,  0,  2,  4])
>> a*b
array([5, 8, 9, 8, 5])
>> a/b
array([ 0.2,  0.5,  1. ,  2. ,  5. ])
```

Mathematical functions such as `np.sin`, `np.cos`, and `np.sqrt` also work element-wise on vectors.

> Hint
>
> Note that the mathematical functions in the `math` module, such as `math.sin` do not work on NumPy arrays. You must use the corresponding functions in the NumPy module, such as `np.sin`.

Vector operations There are many additional functions that can operate on vectors, for example to compute the minimum, maximum, sum, average, and number of elements.

```
>> a = np.array([1, 2, 3, 4, 5])
>> min(a)
1
>> max(a)
5
>> sum(a)
15
>> np.mean(a)
3.0
>> np.size(a)
5
```

The dot product between two vectors can be computed simply as

```
>> a = np.array([1, 2, 3])
>> b = np.array([3, 2, 1])
>> np.dot(a,b)
10
```

Comparison operators and logical indexing It is possible to compare two vectors element-wise

```
>> a = np.array([1.1, 2.2, 3.3, 4.4])
>> b = np.array([3.1, 2.2, 1.3, 0.4])
>> a > b
array([False, False,  True,  True], dtype=bool)
```

This gives a vector of boolean (True/False) values. This vector can be used for *logical indexing*, i.e. to extract a new vector containing the elements of the original vector for which the index is true.

```
>> a[a > b]
array([ 3.3,  4.4])
```

Multiple vectors of boolean values can also be combined using the logical vector operators `&` (and) and `|` (or).

```
>> a = np.array([1.1, 2.2, 3.3, 4.4])
>> b = np.array([3.1, 2.2, 1.3, 0.4])
>> (a > b) | (a < 2)
array([True,  False,  True,  True], dtype=bool)
```

You can also directly create a vector of boolean values similar to how you have created other vectors with identical values

```
>> a = np.zeros(4, dtype=bool)
>> a
array([False, False, False, False], dtype=bool)
>> b = np.ones(4, dtype=bool)
>> b
array([ True,  True,  True,  True], dtype=bool)
```

Error messages When working with vectors, certain operations are not mathematically well-defined, such as trying to access an element beyond the bounds of the vector, or trying to compute the element-wise product of two vectors of different length. This will result in an error message, which in some cases can be difficult to decipher.

```
>> a = np.array([2, 3, 4])
>> b = np.array([1, 2, 3, 4])
>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3
>> a * b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

NumPy arrays are mutable You should be aware that NumPy arrays we use to store vectors behave a bit differently than the variables we have used so far. When a variable stores a NumPy array, it actually stores only a *reference* to the data in the array. Two variables might store a reference to the *same data*, which means that if the data is modified through one variable, it will affect the data in the other variable also: They are said to be *mutable*.

To illustrate this, consider the following examples. In the first example, it should come as no surprise that the variable `b` will contain the value 1 even though the variable `a` is subsequently modified.

```
>>> a = 1  # a gets the value 1
>>> b = a  # b gets the value of a which is 1
>>> a = 99 # a gets the value 99 (this doesn't affect b)
>>> a
99
>>> b
1
```

This is no different for NumPy arrays, when the whole array is assigned a new value

```
>>> a = np.array([1, 2, 3]) # a points to the vector [1, 2, 3]
>>> b = a                   # b points to the same data as a
>>> a = np.array([4, 5, 6]) # a points to a new vector [4, 5, 6] (doesn't affect b)
>>> a
array([4, 5, 6])
>>> b
array([1, 2, 3])
```

However, if two variables reference the same data, modifying the data will affect both variables.

```
>>> a = np.array([1, 2, 3]) # a points to the vector [1, 2, 3]
>>> b = a                   # b points to the same data as a
>>> c = np.copy(a)          # c points to a new vector (copy of a)
>>> a[0] = 99               # The first element of the data pointed to by a (and b) is
                            #  modified, affecting a and b (but not c)
>>> a
array([99,  2,  3])
>>> b
array([99,  2,  3])
>>> c
array([1, 2, 3])
```

As the example also illustrates, one way to avoid this behaviour is to use the `np.copy` function.

Exercise 2B Initializing and indexing vectors

Use the appropriate vector initialization functions and operators to create the following vectors:

1. $\boldsymbol{v}_1 = [3, 7, 1]$ (Hint: Use direct assignment)
2. $\boldsymbol{v}_2 = [0, 0, 0, 0, 0, 0]$ (Hint: Use `np.zeros`)
3. $\boldsymbol{v}_3 = [1, 1, 1]$
4. $\boldsymbol{v}_4 = [1, 2, 3, 4]$
5. $\boldsymbol{v}_5 = [1, 2, 3, 4, 5, 10, 11, 12, 13, 14]$

Use appropriate methods to access the vectors to answer the following questions:

1. Access the first element of the $\boldsymbol{v}_1$ vector.
2. Access the second element of the $\boldsymbol{v}_4$ vector.
3. What happens if you try to access the fourth element of vector $\boldsymbol{v}_3$.
4. How can you change vector $\boldsymbol{v}_4$ to store 10 as the fifth element?
5. How can you change the vector $\boldsymbol{v}_5$ so that it becomes equal to $[3, 7, 1]$?

2B

Exercise 2C Vector operations

Create the vectors $\boldsymbol{v}_1 = [1, 2, 3, 4, 5]$, $\boldsymbol{v}_2 = [3, 4, 5, 6, 7]$, and $\boldsymbol{v}_3 = [1, 1, 1, 1]$. Create a script that outputs the following:

1. The dot product of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$.
   $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = 1 \cdot 3 + 2 \cdot 4 + 3 \cdot 6 + 4 \cdot 7 + 5 \cdot 8 = 85$.
2. Element-wise multiplication of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$.
   $[1 \cdot 3, 2 \cdot 4, 3 \cdot 6, 4 \cdot 7, 5 \cdot 8] = [3, 8, 15, 24, 35]$.
3. The sine function applied to each element in $\boldsymbol{v}_1$.
   $\sin(\boldsymbol{v}_1) = [0.8415, 0.9093, 0.1411, -0.7568, -0.9589]$.
4. The length of $\boldsymbol{v}_1$ which is 5.
5. The dot product of $\boldsymbol{v}_1$ and $\boldsymbol{v}_3$.
   (Hint: What happens when you try to compute this?)

Exercise 2D Logical vector indexing

Create the vectors $\boldsymbol{v}_1 = [4, 2, 1, 2, 5]$ and $\boldsymbol{v}_2 = [-1, 4, 5, -3, 6]$. Use logical indexing to compute a new vector $\boldsymbol{v}_3$ containing the following:

1. The elements of $\boldsymbol{v}_1$ that are less than 3. ($\boldsymbol{v}_3 = [2, 1, 2]$)
2. The elements of $\boldsymbol{v}_2$ that are negative. ($\boldsymbol{v}_3 = [-1, -3]$)
3. The elements of $\boldsymbol{v}_2$ that are greater than 0. ($\boldsymbol{v}_3 = [4, 5, 6]$).
4. The elements of $\boldsymbol{v}_1$ that are greater than 100. There are none, so what happens?
5. The elements of $\boldsymbol{v}_1$ that are greater than the corresponding element in $\boldsymbol{v}_2$. ($\boldsymbol{v}_3 = [4, 2]$)
6. The elements of $\boldsymbol{v}_2$ that are not equal to 5. ($\boldsymbol{v}_3 = [-1, 4, -3, 6]$)
7. The elements of $\boldsymbol{v}_1$ that are greater than the average of the elements in $\boldsymbol{v}_1$. ($\boldsymbol{v}_3 = [4, 5]$)

Solution to selected questions

```
v1 = np.array([4, 2, 1, 2, 5])
v2 = np.array([-1, 4, 5, -3, 6])
```

1. `v3 = v1[v1 < 3]`
3. `v3 = v2[v2 > 0]`
5. `v3 = v1[v1 > v2]`
7. `v3 = v1[v1 > np.mean(v1)]`

Exercise 2E Modifying vectors

Consider the vector $\boldsymbol{v} = [4, 7, -2, 9, 3, -6, -4, 1]$. Which Python command will do the following:

1. Set all negative values to zero. ($\boldsymbol{v} = [4, 7, 0, 9, 3, 0, 0, 1]$)
2. Change the sign of all values. ($\boldsymbol{v} = [-4, -7, 2, -9, -3, 6, 4, -1]$)
3. Set all values that are less than the average to zeros. $\boldsymbol{v} = [4, 7, 0, 9, 3, 0, 0, 0]$
4. Set all negative values to positive. ($\boldsymbol{v} = [4, 7, 2, 9, 3, 6, 4, 1]$)
5. Multiply all positive values by two. ($\boldsymbol{v} = [8, 14, -2, 18, 6, -6, -4, 2]$)
6. Raise all values to the power of 2, but keep their original sign. ($\boldsymbol{v} = [16, 49, -4, 81, 9, -36, -16, 1]$)

Solution to selected questions

1. `v[v < 0] = 0`
3. `v[v < np.mean(v)] = 0`
5. `v[v > 0] = v[v > 0] * 2`

Testing

A very important part of programming is to test if the code functions correctly. Even if your code is very simple, it can be quite difficult to completely understand how it will function in all possible cicumstances. For that reason it is important to have a structured approach to testing your code.

Using assertions

A common way to make sure your code behaves as it should is to use socalled assertions. These are statements that check a condition (a boolean expression) which you know must be true when the program executes. For example, if you have a variable called `speed` which you know must be a positive number, you can add the following assertion statement

```
assert speed > 0
```

When your program executes the assertion statement it will stop if the condition is not true and print out an error message. This allows you to go back and examine the code to find out what happened and fix the bug.

Writing assertion statements is one of the fastest and most efficient methods to testing your code and avoiding bugs.

Unit tests

A unit test is a piece of code that tests a unit of software (such as a single function) to check if it satisfies some requirement. For example, it can check for a given input that the output is as expected. Every time you write a function, we recommend that you write a unit test.

As an example, consider the following function which *attempts* to compute the final velocity, v_f, (of some object) given the initial velocity, v_i, acceleration a, and travelled distance, d, which is governed by the following mathematical formula

$$v_f = \sqrt{v_i^2 + 2 \cdot a \cdot d} \tag{2.3}$$

```
def compute_velocity(velocity_initial, acceleration, distance):
    # Compute final velocity
    velocity_final = math.sqrt(velocity_initial + 2*acceleration*distance)
    return velocity_final
```

There is a bug in the code, but maybe we don't see it immediatly. However, when we write our unit test we will easily spot the bug. Some reasonable unit tests for the above function could be the following:

```
# Unit test of compute_velocity
assert compute_velocity(0, 0, 0) == 0
# No accelleration: Final velocity = initial velocity
assert compute_velocity(10, 0, 0) == 10
assert compute_velocity(5, 0, 5) == 5
# No travelled distance: Final velocity = initial velocity
assert compute_velocity(10, 7, 0) == 10
# Positive acceleration and distance: Final velocity is greater than initial velocity
assert compute_velocity(0, 1, 5) > 0
assert compute_velocity(5, 2, 5) > 5
```

Try running the unit tests. Can you find the bug?

Unit tests and CodeJudge

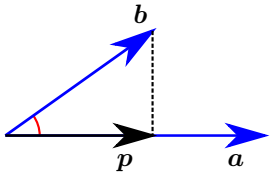
When you submit a solution to an assignment on CodeJudge, a number of unit tests will automatically be run on your code to check that it works correctly. In the real world, when you are faced with the challenge of writing a program to solve a real problem, you will not have access to a nice set of pre-defined unit tests. Therefore it

is important that you learn how to test your code yourself. Here is how you should think about working with CodeJudge in this course.

- Only submit a solution to CodeJudge once you have tested it thoroughly (using unit tests).
- Don't use the unit tests on CodeJudge as the only tests of your code, because in real life you will not have access to any predefined test cases. Learning to program also means learning to test.
- Always aim at testing your code well enough to expect it to pass all tests on CodeJudge in your first submission.
- Don't keep resubmitting to CodeJudge in a trial-and-error process until you pass all the tests. If you don't pass the tests on CodeJudge in your first attempt, you have clearly not tested your solution well enough, and you should go back and reconsider your approach to testing.

Assignment 2F Projection

The *projection* of a vector $\boldsymbol{b}$ onto a vector $\boldsymbol{a}$ is a vector $\boldsymbol{p}$ which contains the vector component of $\boldsymbol{b}$ in the direction of $\boldsymbol{a}$.



The projection can be computed using the following formula

$$\boldsymbol{p} = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|^2} \boldsymbol{a}, \tag{2.4}$$

where $\|\boldsymbol{a}\|$ denotes the magnitude of the vector $\boldsymbol{a}$ (and similar for $\boldsymbol{b}$). The magnitude of the vector can be computed using Pythagoras' formula.

Problem definition

Create a function named `computeProjection` that computes the projection of the vector $\boldsymbol{b} = [1, 1, \dots]$ onto the vector $\boldsymbol{a}$ which is taken as an input to the function. The output must be the projection $\boldsymbol{p}$. Note that $\boldsymbol{b}$ is a vector with all elements equal to one and with the same dimensions as $\boldsymbol{a}$, and that the function should work for vectors of any dimensionality.

Solution template

```
import math
import numpy as np
def computeProjection(a):
  #insert your code
  return projection
```

> **Hint**
>
> Remember that you must import the modules your function needs before defining your function. For example, if you need the `math` and `numpy` modules, you should write the import statements as shown above. In the rest of the exercises we will not explicitly show the import statements in the solution templates.

Input	
`a`	Input vector $\boldsymbol{a}$.
Output	
`projection`	Projection vector $\boldsymbol{p}$.

Example

Consider the vectors $\boldsymbol{a} = [2, -1]$ and $\boldsymbol{b} = [1, 1]$. The projection of $\boldsymbol{b}$ on $\boldsymbol{a}$ can be computed as

$$\text{projection} = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|^2} \boldsymbol{a} = \frac{2 \cdot 1 + (-1) \cdot 1}{2^2 + (-1)^2} [2, -1] = [0.4, -0.2]. \tag{2.5}$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(computeProjection(np.array([2, -1])))`	`[ 0.4 -0.2]`

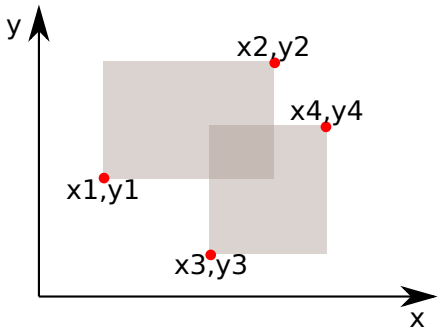
Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

2F

Assignment 2G Box area

Two rectangular boxes are defined by the coordinates of their lower left and upper right corners as seen below.



The area of the two boxes are given by

$$A_1 = (x_2 - x_1)(y_2 - y_1), \qquad A_2 = (x_4 - x_3)(y_4 - y_3). \tag{2.6}$$

The area of the intersection between the two boxes is given by

$$A_o = \max\big(0, \min(x_2, x_4) - \max(x_1, x_3)\big) \cdot \max\big(0, \min(y_2, y_4) - \max(y_1, y_3)\big). \tag{2.7}$$

Problem definition

Create a function named `boxArea` that takes as input the coordinates defining the two boxes as a vector $[x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4]$ as well as a string `area` that specifies what to compute (written exactly as below). The function must return the computed area.

`area`	Area to compute
`Box1`	Area of the first box.
`Box2`	Area of the second box.
`Intersection`	Area of the overlap between the boxes.
`Union`	The sum of the areas of the two boxes minus the area of the overlap.

Solution template

```
def boxArea(boxCorners, area):
    # Insert your code here...
    # Use an if-elif statement to choose between the different areas to compute.
    if area == "Box1":
        # Insert code to compute area of box one
        # A = ...
    elif area == "Box2":
        # Insert code to compute area of box two
        # A = ...
    elif area == "Intersection":
        # Insert code to compute area of intersection
        # A = ...
    elif area == "Union":
        # Insert code to compute area of union
        # A = ...
    return A
```

Input	
`boxCorners`	Corners of the boxes (vector of length 8).
`area`	Which area to compute (string).
Output	
`A`	The computed area (number).

Example

Consider the following coordinates: $x_1 = 5, x_2 = 20, x_3 = 14, x_4 = 25, y_1 = 12, y_2 = 23, y_3 = 5, y_4 = 17$. The input `boxCorners` would be the vector $[5, 20, 14, 25, 12, 23, 5, 17]$. If the input string `area` is equal to `intersection` the area of the overlap must be computed: $A_o = \max[0, \min(20, 25) - \max(5, 14)] \cdot \max[0, \min(23, 17) - \max(12, 5)] = \max[0, 20 - 14] \cdot \max[0, 17 - 12] = 6 \cdot 5 = 30$.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(boxArea(np.array([5,20,14,25,12,23,5,17]), "Intersection"))`	`30`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

2G

Optional challenge 2H Sudoku row

Sudoku is a number puzzle where the objective is to fill a 9-by-9 grid with numbers between 1 and 9 so that each row, each column, and each 3-by-3 block contains all the numbers from 1 through 9. Sudokus are difficult to solve, but when a row misses only a single number, it is a simple yet tedious task to fill in the blank.

9	4		1	5	7	2	3	8

Problem definition

Create a function named `fillSudokuRow` that takes as an input a Sudoku row with one missing entry (marked by a value of zero), and returns the row with the entry filled in correctly so that all numbers from 1 to 9 appear once in the row.

> Hint
>
> There is a quick and simple way to solve this exercise, using only the techniques you have learned in this module.

Solution template

```
def fillSudokuRow(sudokuRow):
  # Insert your code here
  return sudokuRow
```

Input	
`sudokuRow`	Sudoku row with one missing entry set to zero (vector)
Output	
`sudokuRow`	Sudoku row with missing entry filled in (vector)

Example

In the Sudoku row

$$[9, 4, 0, 1, 5, 7, 2, 3, 8]$$

the third value is marked as missing, and should be filled in. Since the number 6 does not appear in the row, the missing value should be replaced with 6. Thus your program should output the following Sudoku row,

$$[9, 4, 6, 1, 5, 7, 2, 3, 8].$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(fillSudokuRow(np.array([9, 4, 0, 1, 5, 7, 2, 3, 8])))`	`[9 4 6 1 5 7 2 3 8]`

Hand in on CodeJudge

In order to validate you can either upload a file containing your implementation of the function or directly copy the code into codejudge.

2H

Selection statements

Aims and objectives

After working through this exercise you should be able to:

- Use *selection* and *branching* statements:
  - Use an *if-statement* to execute some code only if some condition is true.
  - Use an *if... else-statement* to execute different code depending on a condition.
  - Use nested if-statements and *if... elif... else-statements* to execute different parts of the code depending on multiple conditions.
- Use the basic *comparison operators* to:
  - Check if two variables or values are (not) equal.
  - Check if one numeric variable or value is greater than, greater than or equal, less than, or less than or eqaul to another numeric variable or value.
- Use the basic *logical operators* including conjunction (and), disjuction (or), and negation (not).
- Use logical and comparison operators to define *conditions* for selection and brancing statements, in which you:
  - Compare numeric variables, text strings, and logical values.
  - Combine multiple comparisons using (and) and (or).

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 5.1–5.7.

Video: If-statements

Using selection statements

Selection statements are sometimes referred to as code branching, because you branch your program into execution of different sequences of code based on one or more conditions. The most simple selection statement is the *if-statement*, which will run one or more lines of code if the condition is true, or otherwise do nothing (not run the lines of code).

```
if condition:
  print('Condition is true')
```

In the example above, the line of code which prints out `Condition is true` will only be printed if the variable `condition` evaluates to true. Usually, in place of the `condition` you would write some *expression* which evaluates to True/False, e.g., a comparison statement such as `x > 0`. To explicitly write the values true and false, you simply write `True` or `False`. Notice that even though the condition is not a True/False value—for example if it is a number or a string—Python will try to interpret the condition as a True/False value if possible. This can be a source of potential problems, so it is recommended to only use True/False expressions as conditions.

If you wish to run one branch of code if a condition is true and another branch if it is not true, you can use an *if... else-statement*.

```
if condition:
  # Do something if condition is true
  print('Condition is true')
else:
  # Do something else if condition is false
  print('Condition is false')
```

To improve the readability of your code, you should always use parantheses when combining more than two statements as a condition. For instance, it is not so easy to read the following code if-statement:

```
if height<10 and width>20 or width>10 and height<20
```

Adding parentheses makes the expression easier to read:

```
if (height<10 and width>20) or (width>10 and height<20)
```

Exercise 3A True or false

Does each of the following bits of code print out `Condition is true` or `Condition is false`? Answer by looking at the code, and verify your answers with Python.

1. ```
   if math.pi < 3:
     print('Condition is true')
   else:
     print('Condition is false')
   ```

2. ```
   x = 10
   if x < 5:
     print('Condition is true')
   else:
     print('Condition is false')
   ```

3. ```
   x = 5
   if math.pi > 3 and x == 5:
     print('Condition is true')
   else:
     print('Condition is false')
   ```

4. ```
   if math.pi < 3 or 5 == 5:
     print('Condition is true')
   else:
     print('Condition is false')
   ```

Try also the following values as conditions. These values are not True/False, but you can see if and how they are converted to True/False values when used as conditions.

5. `0` (the number zero)
6. `44` (some non-zero number)
7. `""` (an empty string)
8. `"hello"` (some non-empty string)
9. `"false"` (the string `false`)
10. `np.array([0, 1])` (a vector—do its values matter?)

3A

Exercise 3B Logical expressions

Evaluate the the following logical expressions (note that all the expressions have the value true or false). Evaluate the expressions on paper or in your mind first, and then write Python code that evaluates the expression to verify your answers.

1. $9 > 3 \cdot 3$.
2. $9 \geq 3 \cdot 3$.
3. $(6 + 3)$ is equal to 3^2.
4. $\frac{12}{5}$ is greater than $\frac{17}{5}$.
5. Either (2 is greater than 3) or (3^2 is not equal to 9).
6. 323 is not equal to (17 times 19).
7. The cosine of (3 times π) is a negative number.

Solution to selected questions

1. `9 > 3*3`
3. `6+3 == 3**2`
5. `(2 > 3) or (3**2 != 9)`
7. `math.cos(3*math.pi) < 0`

Comparison pitfalls

When comparing variables there are some common pitfalls to be aware of.

Intervals In mathematical notation, we often write intervals as for example $0.5 \leq x \leq 1.5$. If you want to check if a variable `x` is in the interval between 0.5 and 1.5, you might be tempted to write something like `if 0.5 <= x <= 1.5`. However, this will not work as you might expect. The correct way to check if a variable is within an interval is to check the upper and lower boundary separately and combine them, such as `if (0.5 <= x) and (x <= 1.5)`

Comparing vectors When you compare two vectors of equal length in Python, the `==` operator compares the vectors element by element and returns a vector with the results of the comparisons. If you want to check if all elements in two vectors are equal, you can use the `all` function. Alternatively you can use the `np.array_equal` function (see the help to learn more about this). If the vectors compared with the `==` do not have the same length, the comparison will return False.

Comparing decimal numbers Decimal numbers are represented on the computer with limited precision. Therefore, some comparisons which analytically should be true, migth be false because of round-off errors. For example, the expression `math.sqrt(2)**2 == 2` evaluates to *false* because the square root of two is not accurately represented in the computer. The expression `50.0**50.0 + 1 == 50.0**50.0` evaluates to *true*, because the number 50^{50} is so big compared to the $+1$ that it is rounded off. Instead of checking if two decimal numbers are equal, it can be more safe to check if their absolute difference is less than some small number (relative to their magnitude).

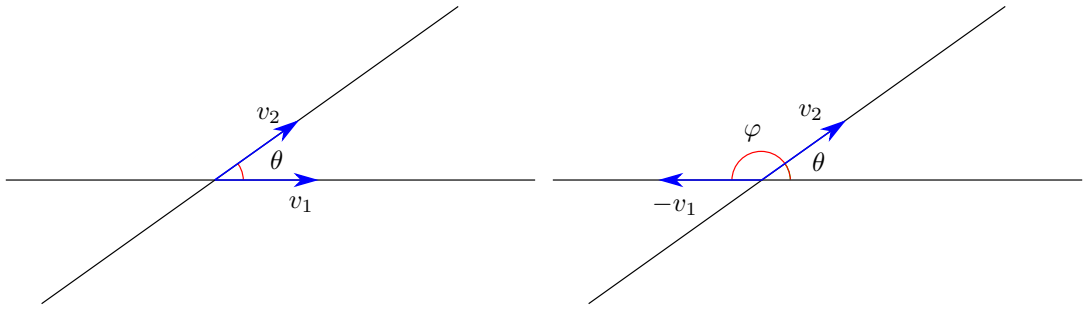
Comparing strings Sometimes, when you compare two strings, your wish to be insensitive to the case, e.g., such that the strings `Hello` and `hello` are considered equal. One way to achieve this is to first convert the strings to lower (or upper) case before they are compared. You can use the notation `str.lower()` where `str` is a string to convert a string to lower case.

Assignment 3C Angle between lines

A line through the origin can be represented by a unit vector pointing in the direction of the line. The angle between two such lines can be computed as

$$\measuredangle v_1, v_2 = \cos^{-1}(v_1 \cdot v_2), \tag{3.1}$$

i.e. as the inverse cosine (also known as arccos) of the dot product of v_1 and v_2 which are unit vectors representing the two lines. If v_1 is a unit vector representing a line, $-v_1$ will also be a unit vector representing the same line. When computing the angle between two lines, there are thus two correct answers: The acute angle θ and the obtuse angle φ.



Problem definition

Write a function that takes as input two unit vectors v_1 and v_2 representing two lines, and computes the *acute* angle between the lines measured in radians, $\theta \in \left[0, \frac{\pi}{2}\right]$.

Solution template

```
def acuteAngle(v1, v2):
  # Insert your code here
  return theta
```

Input	
`v1`	Unit vector determining v_1
`v2`	Unit vector determining v_2
Output	
`theta`	The acute angle between the lines, $\theta \in \left[0, \frac{\pi}{2}\right]$.

Example

Consider two lines represented by the following unit vectors: $v_1 = \left[-\frac{4}{5}, \frac{3}{5}\right]$ and $v_2 = \left[\frac{20}{29}, \frac{21}{29}\right]$. The angle between the lines are given by

$$\measuredangle v_1, v_2 = \cos^{-1}(v_1 \cdot v_2) = \cos^{-1}\left(-\tfrac{4}{5} \cdot \tfrac{20}{29} + \tfrac{3}{5} \cdot \tfrac{21}{29}\right) = \cos^{-1}\left(-\tfrac{17}{145}\right) = 1.688 \text{ radians} \tag{3.2}$$

Since this angle is obtuse (it is greater than $\frac{\pi}{2}$) we determine the acute angle to be $\theta = \pi - 1.688 = 1.453$.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(acuteAngle(np.array([-4.0/5.0, 3.0/5.0]), np.array([20.0/29.0, 21.0/29.0])))`	`1.453284681363451`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

3C

Assignment 3D Piecewise function

On the surface of the earth the graviational pull is approximately $g_0 = 9.82$ [m/s^2]. If you move up towards space the gravitational pull declines with the height, h, above the ground according to the equation

$$g_{\text{height}}(h) = g_0 \cdot \frac{R^2}{(R+h)^2} \tag{3.3}$$

where $R = 6.371 \cdot 10^6$ [m] is the average radius of the earth. Similarly if you dig yourself into the ground there is less mass of the earth underneath you and the graviational pull is weaker and declines as a function of the depth, d, of the hole you are digging:

$$g_{\text{depth}}(d) = g_0 \left(1 - \frac{d}{R}\right) \tag{3.4}$$

This is a rough approximation as the earth is not a ball; nor is the density of the earth constant. For more details, visit the wikipedia page: en.wikipedia.org/wiki/Gravity_of_Earth. These two equations can be combined to describe the graviational pull at some distance, $x = R + h = R - d$, from the center of the earth.

$$g_{\text{distance}}(x) = \begin{cases} g_0 \cdot \dfrac{R^2}{x^2} & \text{if } R \leq x \\ \\ g_0 \cdot \dfrac{x}{R} & \text{if } 0 \leq x < R \end{cases} \tag{3.5}$$

Problem definition

Write a function that, given the distance x to the center of the earth, computes the graviational pull of the earth.

Solution template

```
def gravitationalPull(x):
  # Insert your code here
  return g
```

Input	
d	Distance to the center of the earth
Output	
g	The gravitational pull.

Example

Suppose we are given $x = 1.78 \cdot 10^6$ [m] as input to the function. Since $1.78 \cdot 10^6 < 6.371 \cdot 10^6$, inserting $1.78 \cdot 10^6$ into equation 3.5 yields

$$g_{\text{distance}}(1.78 \cdot 10^6) = g_0 \cdot \frac{1.78 \cdot 10^6}{R} = 9.82 \cdot \frac{1.78 \cdot 10^6}{6.371 \cdot 10^6} = 2.7436 \text{ [m/s}^2\text{]} \tag{3.6}$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(gravitationalPull(1.78e6))`	`2.7436195259770835`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

Discussion and futher analysis

What will happen if you give a vector of distances as input to your function? Probably it will not work, because the function expects a single number as input. How could you write the function such that it outputs a vector of gravitational pull when given a vector of distances as input? Think about how this could be implemented using vector indexing operations.

If you change your function so that it works correctly with vector input and output, you can try out making a plot of the function using the following code:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 10e6, 1e4)
plt.plot(x, gravitationalPull(x))
plt.show()
```

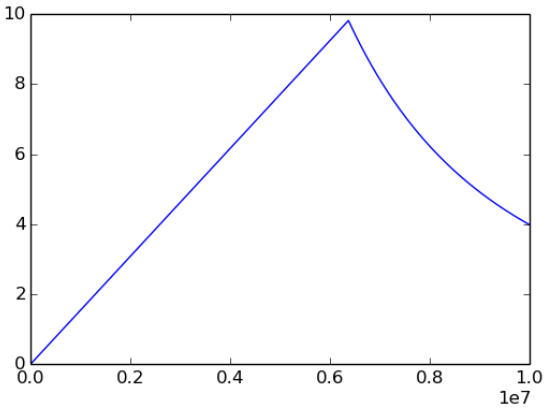


Figure 3.1: A plot of the gravitational pull from the center of the earth to a distance of 10 000.

Assignment 3E Acidity

"In chemistry, pH is a measure of the acidity or basicity of an aqueous solution. Solutions with a pH less than 7 are said to be acidic and solutions with a pH greater than 7 are basic or alkaline. Pure water has a pH very close to 7." [Wikipedia] The acidity of a solution can be categorized according to the following scale:

pH	Category
0–2	Strongly acidic
3–5	Weakly acidic
6–8	Neutral
9–11	Weakly basic
12–14	Strongly basic

Problem definition

Create a function that converts a pH value to the corresponding category. If the pH is between two categories, it must be assigned to the strongest (acidic or basic) category of the two. If the pH is a number outside the scale the string `pH out of range` must be returned.

Solution template

```
def pH2Category(pH):
  # Insert your code here
  return category
```

Input	
`pH`	pH value (real scalar).
Output	
`category`	Acidity category (string).

Example

The pH of lemon juice is 2.3. Since this is between the *Strongly acidic* and *Weakly acidic* categories, it must be assigned to *Strongly acidic*. Thus, the string `Strongly acidic` should be the output of the function.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(pH2Category(2.3))`	`Strongly acidic`

Hand in on CodeJudge

This assignment must be handed in on CodeJudge.

3E

Optional challenge 3F Football goal tracker

The 2013–2014 Premier League season was the first time the Hawk-Eye system was used for tracking the ball to detect when a goal was scored. The system uses a number of high-performance cameras to track the ball from different angles.

We might imagine a simple version of the system that, once the ball is kicked, tracks whether or not the ball would pass the goalline if it continued in a straight line. Suppose the system is already able to compute the point where the ball is kicked and the direction vector of the kick.

The standard measurement of a football field is 105 [m] × 68 [m] and the goal is 7.32 [m] wide. Suppose the ball is shot from a point $p = [x, y]$ on the field and goes in a straight line in some direction given by a vector $v = [v_x, v_y]$. (The magnitude of the vector indicates the speed of the ball, which we will not use in this exercise.)

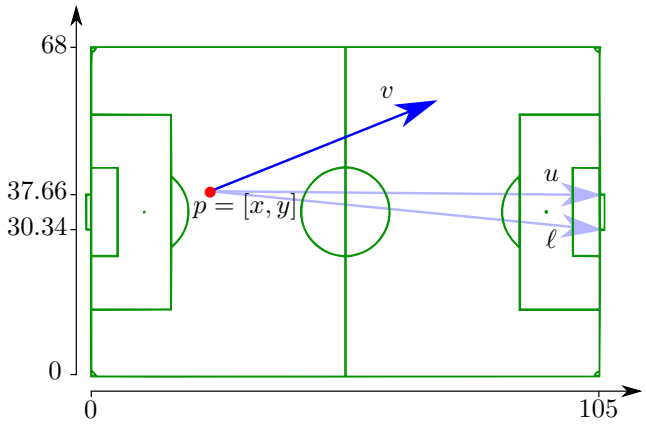


Figure 3.2: A football field.

Problem definition

Create a function that tells whether or not the ball will pass the goalline (in either of the two goals) if it continues from the initial position in a straight line along the direction vector.

Solution template

```
def computePassesGoalLine(point, directionVector):
  # Insert your code here
  return score
```

Input	
`point`	The (x, y) coordinates of the initial position of where the football is kicked from (vector).
`directionVector`	The vector describing the direction in which the football is kicked.
Output	
`score`	A boolean telling whether or not the ball will pass the goalline.

Example

Assume that the position of the ball is given by the point $p = [x, y] = [30, 20]$ and the direction is given by the vector $v = [v_x, v_y] = [10, 2]$.

Firstly, we note that since the x-component of the direction vector is positive, in this case the ball can only pass the goal line in the goal on the right. The goal posts of the goal on the right are at the positions $[105, 30.34]$ and $[105, 37.66]$.

The ball will cross x-coordinate $x_{\text{goal}} = 105$ at the right at position $p + \alpha v$ where α is a number denoting how far we should travel along v. Since we must have $x + \alpha v_x = 105$ we can compute α as

$$\alpha = \frac{x_{\text{goal}} - x}{v_x} = \frac{105 - 30}{10} = 7.5. \tag{3.7}$$

The y-coordinate where the ball passes the goal line is then given by

$$y_{\text{goal}} = y + \alpha v_y = 20 + 7.5 \cdot 2 = 35. \tag{3.8}$$

Then the condition to see if the ball ends up in the goal is equivalent to testing if

$$30.34 < y_{\text{goal}} < 37.66, \tag{3.9}$$

which is true for $y_{\text{goal}} = 35$ so the return value `score` should be set True.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(computePassesGoalLine(np.array([30, 20]), np.array([10, 2])))`	`True`

Hand in on CodeJudge

This challenge can be be handed in on CodeJudge.

Looping

Aims and objectives

After working through this exercise you should be able to:

- Use loop statements to repeatedly execute code.
  - Use *while loops* to execute code repeatedly as long as a condition is true.
  - Use *for loops* to execute code repeatedly for a fixed sequence of iterations.
- Use a *break statement* to terminate the execution of a loop, if some condition is true.
- Use *nested loops*, i.e., a loop inside another loop.
- Use *vectorized computations*.
  - Determine whether a loop can be vectorized and describe why some loops can not be vectorized.
  - Convert code using a loop into a vectorized computation and the other way around.
- Create functions that accept *multiple inputs*.
- Display formatted output.

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 4.2 + 7.

Video: Loops
Video: Using the debugger

For- and while-loops

If you want to run one or more lines of code multiple times, you can use a loop. Sometimes you know in advance how many times you would like the loop to run—in that case you most often use a *for-loop*. Other times, you want to run the loop an unknown number of times, stopping only when some criterion is met dependeing on the computations inside the loop—in that case you most often use a *while-loop*.

For-loop In the for-loop you specify in advance how many times you would like to loop to run. For example, the following loop will print out the numbers 0 to 3 on the screen:

```
for i in range(4):
  print(i)
```

> Hint
>
> Make sure to read the documentation for the `range` function to understand what it does.

In effect, this corresponds to the code

```
i = 0
print(i)
i = 1
print(i)
i = 2
print(i)
i = 3
print(i)
```

The range you loop over does not have to be a fixed constant: It could also depend on other variables. For example, the following code will print out the numbers from `minNumber` to `maxNumber` on the screen:

```
minNumber = 10
maxNumber = 19
for i in range(minNumber, maxNumber+1):
  print(i)
```

While-loop In the while-loop you must specify a condition (a True/False expression) and the loop will continue to repeat as long as the condition is True. For example, the following while-loop will iteratively sum up the numbers $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$ one by one and print out the sum, as long as sum is less than 3. Thus it will print out $1, 1.5, 1.8333, 2.0833, \ldots$ etc.:

```
theSum = 0
i = 1
while(theSum < 3):
  theSum = theSum + 1.0 / i
  print(theSum)
  i = i + 1
```

The while-loop first examines the condition, and since it is True it runs the body of the loop once. Then it examines the condition again, and since it is still True it runs the body of the loop again. It continues doing this until the condition is False, at which point it stops looping and continues to run the code following the loop. In the example above there is no more code after the loop, so it simply stops. Of course, the condition (in the example `theSum < 3`) must somehow depend on something which is computed inside the loop, so that it changes as the loop runs.

Try running the while-loop in Python. Does it work as you would expect? If you expected that the loop would print out some numbers which are all less than 3, you might be surprised that it actually prints out the numbers $1, 1.5, 1.8333, 2.0833, 2.2833, 2.45, 2.5929, 2.7179, 2.829, 2.929, 3.0199$. The last number printed is greater than 3,

because the loop will always be run *once more* as long as `theSum < 3`. After the second-to-last iteration, `theSum` is equal to 2.929 (still less than 3) and thus the body of the loop is once run again, increasing `theSum` to 3.0199 and printing it out. After this iteration, `theSum` is greater than 3, and the loop stops.

For-loop or while-loop? Even though both for-loops and while-loop are quite similar, there is an important semantic difference between the two types of loops:

While-loops are most often used to run for an indefinite number of iterations, i.e. in the case when the number of iterations is not known in advance. For example, to read each line from a file, until there are no more lines, or to run some iterative algorithm until a convergence criteria is satisfied.

For-loops are most often used when there is a known number of iterations, for example to process each element in a list, where the length of the list is known.

Breaking a loop It is possible to stop a loop (both for- and while-loops) prematurely using a `break` statement. Usually the `break` statement will be inside an `if` statement, so that the loop will stop running if some condition is satisified. For example, the following while-loop breaks if `theSum` is greater or equal to 3, before printing out the value. Thus, it will only print out the numbers that are less than 3.

```
theSum = 0
i = 1
while(True):
  theSum = theSum + 1.0 / i
  if (theSum >= 3):
    break
  print(theSum)
  i = i + 1
```

In general, it is not recommended to use `break` statements if they can be avoided, because they often lead to code that is more difficult to read and understand. For example, the code above can be rewritten as:

```
theSum = 1
i = 1
while(theSum < 3):
  print(theSum)
  i = i + 1
  theSum = theSum + 1.0 / i
```

Which version do you find most easy to read?

Loop pitfalls

When using for-loops and while-loops, there are some common pitfalls to be aware of:

Return statement in the loop Be careful that you do not accidentally put a return statement inside a loop. If you do that, the loop will only run once, and the return when it reaches the return statement.

Off-by-one error Especially when using a while loop, make sure to double check that your loop does not run one time too much or one time too little, which is a very common mistake to make.

Infinite loop In a while-loop, if the condition never becomes *false*, the loop will run forever. If this happens, the computer program will be stuck in an infinite loop, and you have to manually stop the program.

Loops and vectorized computation

Loops generally serve to remove code redundancy, and they improve code readability if used properly. In general, if the same code needs to be run multiple times, it is desirable to use a loop rather than having multiple identical or almost-identical statements.

Although loops are powerful when used the right way, be aware that they can sometimes be computationally inefficient. This is particularly true for nested loops (loops inside loops). It is therefore important that you do not use loops blindly: Use them when you have to and/or when there is a gain in terms of code readability and reduction of redundancy.

When working with vectors you have learned how you can operate on multiple elements at the same time. Often a computation which can be written using a loop can also be written using operations on one or more vectors. Doing repeated computation using vectors rather than loops is often referred to as *vectorization* of the code. In general, you shoul use vectorization (instead of loops) if it is possible and if it leads to better code. Often but not always, vectorization leads to code that is more intuitive and easier to understand, and usually vectorization is more efficient than looping.

For example, the following code operates on a vector `x` containing the numbers $-2, -1, 0, 1, 2$. It uses a loop to set all the negative numbers equal to zeros and multiply all the positive numbers by two, producing as an output a vector with the numbers $0, 0, 0, 2, 4$:

```
x = np.array([-2, -1, 0, 1, 2])
for i in range(5):
  if(x[i] < 0):
    x[i] = 0
  else:
    x[i] = x[i] * 2
```

Using vectorized computation, the same result could have been achieved more efficiently and in a more intuitive way with the following code:

```
x = np.array([-2, -1, 0, 1, 2])
x[x < 0] = 0
x = x * 2
```

Use the following guidelines when you need to choose between for-loops, while-loops, and vectorization:

- Vectorize for efficiency when it is appropriate.
- When you cannot vectorize and you know how many times you need to loop, use a for-loop.
- When you cannot vectorize and you do not know how many times your loop should run, use a while-loop.

Displaying formatted output

To display strings or results of expressions on the screen, you can use the `print` function. Very often we need to display more complicated formatted output on the screen. To do this, you can use the `str.format` function. For example the code

```
x = math.pi
print("The cos of {:f} is {:f}".format(x, math.cos(x)))
```

will print out

```
The cosine of 3.141593 is -1.000000.
```

The placeholder `{:f}` denotes that a decimal number will be inserted in the string. See the Python documentation to learn more about the syntax.

How can you modifiy the code, so that the values are printed with 3 decimals?

Exercise 4A Repeated printing

1. Write a script that prints `I love programming!` 10 times.
2. Write a script that uses a for-loop to print a sequence of numbers, x_1 to x_{10}, where $x_1 = 2$ and the subsequent numbers are computed as $x_i = 2 \cdot x_{i-1}$. Check that your program prints the sequence `2 4 8 16 32 64 128 256 512 1024`.
3. Write a script uses a for loop to print out the following:
   ```
   The square root of 1 is 1.0000
   The square root of 2 is 1.4142
   The square root of 3 is 1.7321
   ⋮
   The square root of 10 is 3.1623
   ```
   Make sure that the square roots are printed with four decimals.
4. Write a script that uses a loop to print
   ```
   The train will leave at 13:36 tomorrow
   The train will leave at 13:56 tomorrow
   The train will leave at 14:16 tomorrow
   ⋮
   The train will leave at 17:16 tomorrow
   ```
   You may assume that trains leave every 20 minutes.

> **Hint**
>
> You can consider making one variable to represent the hours, and another variable to represent the minutes. Then you can use `str.format` to print a formatted string:
>
> ```
> h = 13
> m = 36
> print("The train will leave at {:0d}:{:0d} tomorrow".format(h, m))
> ```
>
> Make sure you know what the format string placeholder `{:0d}` means. Next, you can write a loop around the print statement, so that the string is printed out multiple times. Within the loop, you can then add some lines of code that increase the minutes by 20 at each iteration. If the minutes exceed 60, then the hours should increase by 1, and the minutes should decrease by 60.

Exercise 4B Power series approximation

Many important mathematical functions and constants can be represented as power series. For example, $\pi = 3.14\ldots$ can be represented by the infinite series

$$\pi = 4 \cdot (\tfrac{1}{1} - \tfrac{1}{3} + \tfrac{1}{5} - \tfrac{1}{7} + - \cdots). \tag{4.1}$$

Since we cannot write a computer program to evaluate the infinitely many terms, we might approximate π by truncating the series, i.e., summing only terms 0 through N,

$$\pi \approx 4 \cdot \sum_{n=0}^{N} \frac{(-1)^n}{2n+1}. \tag{4.2}$$

Write a script that approximates π by evaluating Eq. (4.2) with $N = 100$.

1. Write the script using a for-loop.
2. Write the script using vectorized computation.

Self-check

With $N = 100$ the approximation is $\pi \approx 3.1515$.

Discussion and futher analysis

- Which implementation (using a loop or vectorized code) do you think is easiest to understand and implement?
- Try running your code with $N = 1\,000\,000$ or maybe $N = 10\,000\,000$ (it might take some time). Is there a significant speed difference between the two implementations?

Solution

Using a for-loop:

```
N = 100
pi_approximation = 0
for n in range(N+1):
   pi_approximation = pi_approximation + 4*(-1)**n / (2*n + 1)
```

Using vectorized computation:

```
n = np.arange(101)
pi_approximation = 4*np.sum((-1)**n / (2*n + 1))
```

4B

Exercise 4C Square roots

The square root of any positive number can be found by a simple iterative procedure that takes an initial guess and refines it. To find the square root of a such that $x = \sqrt{a}$, you start by initializing x to $\frac{a}{2}$. Then you repeatedly update x by replacing it by $\frac{x+\frac{a}{x}}{2}$. As x is updated it will be a better and better approximation of the square root of a.

Problem definition

Write a program that approximates the square root as described above. How many updates are required to get the first five significant digits correct when approximating $\sqrt{2}$?

Self-check

After three updates the result is correct to more than five significant digits.

4C

Assignment 4D Fermentation rate

When a nutrient solution is seeded with yeast, the yeast will grow and ferment sugar into alcohol and carbon dioxide. You are conducting a series of experiments to determine the rate of fermentation, r, and have measurements of the fermentation rate measured in $\frac{\text{gram}}{\text{liter}\cdot\text{day}}$ for a large number of identical experiments. To get a precise estimate of the fermentation rate, you want to compute the average of all the measured rates; however, you notice that some of the measurements are clearly erroneous. You decide to exclude all invalid measurements, which you define as measurements that are outside the range $\ell < r < u$.

Problem definition

Write a program that computes and returns the mean fermentation rate, taking only the valid measurements into account.

Solution template

```
def fermentationRate(measuredRate, lowerBound, upperBound):
  # Insert your code here
  return averageRate
```

Input	
`measuredRate`	Measured fermentation rates (list)
`lowerBound`	Lower bound for valid measurements, ℓ (scalar)
`upperBound`	Upper bound for valid measurements, u (scalar)
Output	
`averageRate`	Average fermentation rate of valid measurements (scalar)

Example

Consider the following measured fermentation rates

$$20.1 \quad 19.3 \quad 1.1 \quad 18.2 \quad 19.7 \quad 121.1 \quad 20.3 \quad 20.0$$

Setting the lower and upper bound to $\ell = 15$ and $u = 25$ the measured rates of 1.1 and 121.1 are invalid, and the remaining 6 measurements are valid. Thus, the mean can be computed as

$$\frac{20.1 + 19.3 + 18.2 + 19.7 + 20.3 + 20.0}{6} = 19.6$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(fermentationRate(np.array([20.1, 19.3, 1.1, 18.2, 19.7, 121.1, 20.3, 20.0]), 15, 25))`	`19.6`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

Discussion and futher analysis

This problem can either be solved using a loop or using vectorized computation. How did you solve the problem, and why?

Assignment 4E Bacteria growth

It has been discovered that a population of bacteria grows according to the following law: If there are n_t bacteria at time t then one hour later at time $t+1$ there will be

$$n_{t+1} = \left(1 + \alpha \cdot \left[1 - \frac{n_t}{K}\right]\right) \cdot n_t$$

where α is a positive number that controls the growth rate and K is a positive number that defines the capacity, i.e., the maximum number of bacteria in the population.

Problem definition

Write a program that simulates the bacteria growth hour by hour and stops when the number of bacteria exceeds some fixed number, N. Your program must return the time t at which the population first exceeds N. Even though the actual number of bacteria is really a whole number (integer) your program must work with n_t as a decimal number (real), i.e., you should not round the numbers.

Solution template

```
def bacteriaGrowth(n0, alpha, K, N):
  # Insert your code here
  return tN
```

Input	
`n0`	Initial number of bacteria (scalar)
`alpha`	Growth rate (scalar)
`K`	Capacity (scalar)
`N`	Final population size (scalar, `n0<N<K`)
Output	
`tN`	Time t at which population size exceeds N (scalar)

Example

Starting with $n_0 = 100$ bacteria and setting $\alpha = 0.4$ and $K = 1000$, let us find out when the population exceeds $N = 500$. Running the simulation, we get the following sequence of population sizes

n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8
100	136	183	242.8	316.3	402.9	499.1	599.1	695.2

Thus, the populations first exceeds $N = 500$ at time $t_N = 7$, the program should return 7.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(bacteriaGrowth(100.0, 0.4, 1000.0, 500.0))`	`7`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

Assignment 4F Removing incomplete experiments

You are working on a data set from a series of experiments, each of which consists of three parts. The experiments have been performed in a randomized order. Each experiment has been assigned an experiment-number and a part-number, which are joined into one decimal number called an id-number. The id-number is formed by separating the experiment-number and part-number by a decimal point. For example, the experiment number 17 part 3 has been assigned the id-number 17.3. Note, that you can compute the experiment-number from the id-number by rounding down to the nearest integer.

You notice that due to errors, for some of the experiments all three parts have not been completed. For further analysis, you need to exclude all experiments where one or more parts are not available. You can safely assume that if there are 3 id-numbers with the same experiment-number, the experiment is complete.

Problem definition

Create a function that takes as an input a vector of id-numbers and returns a vector of id-numbers where all incomplete experiments have been removed. The id-numbers that are not removed must remain in the same order as in the original vector.

Solution template

```
def removeIncomplete(id):
  # Insert your code here
  return idComplete
```

Input	
`id`	Id-numbers (vector of decimal numbers)
Output	
`idComplete`	Id-numbers of complete experiments (vector of decimal numbers)

Example

Consider the following id-numbers:

1.3 2.2 2.3 4.2 5.1 3.2 5.3 3.3 2.1 1.1 5.2 3.1

In experiment 1, part 2 is missing, and in experiment 4, parts 1 and 3 are missing. Thus, experiment 1 and 4 are incomplete. After removing the incomplete experiments, the result should be:

2.2 2.3 5.1 3.2 5.3 3.3 2.1 5.2 3.1

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

`import numpy as np`
`print(removeIncomplete(np.array([1.3, 2.2, 2.3, 4.2, 5.1,`
`  3.2, 5.3, 3.3, 2.1, 1.1, 5.2, 3.1])))`
	`[ 2.2  2.3  5.1  3.2  5.3  3.3  2.1  5.2  3.1]`

Hand in on CodeJudge

This assignment must be handed in on CodeJudge.

4F

Optional challenge 4G Cluster analysis

Using an image sensor on a satellite, you can measure the reflectance at the ground level in a narrow band of the near-infrared frequency spectrum. It is known that this reflectance is highly sensitive to the type of vegetation; however, there is some measurement noise and the overall magnitude of the reflectance also varies with the time of day, sensor drift, etc. You want to create an automated system that can distinguish between two types of vegetation, trees and grass, which are known to have relatively high and low reflectance respectively.

To distinguish between trees and grass, you decide to use *cluster analysis*. "Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters)." [Wikipedia] In particular, you will use the following algorithm (called k-means) to cluster the reflectance data into two groups.

The clustering algorithm

Given a set of N measurements, $(r_1, r_2, \ldots, r_N)$, we will initially assign the odd-numbered measurements to class 1 and the even numbered measurements to class 2. Then the following two steps are repeated:

- *Update step:* Compute the mean value (average) of the measurements within each cluster.
- *Assignment step:* Assign each measurement to the cluster with the closest mean value. In case of a tie, assign the measurement to cluster 1.

Repeat the above steps until the cluster assignments do not change. It can not be determined in advance how many steps will be needed before the clustering assignment stabilizes.

Problem definition

Create a function that takes as an input a vector of reflectance measurements and returns a vector of cluster assignments computed using the algorithm described above.

Solution template

```
def clusterAnalysis(reflectance):
  # Insert your code here
  return clusterAssignments
```

Input	
`reflectance`	Reflectance measurements (list of decimal numbers)
Output	
`clusterAssignments`	Final cluster assignments (list of numbers, 1 or 2)

Example

Consider the following set of reflectance measurements where the color and style of the number indicates the initial cluster assignment,

1.7 *1.6* **1.3** *1.3* **2.8** *1.4* **2.8** *2.6* **1.6** *2.7.*

In the update step, the two mean values can be computed as

$$m_1 = \frac{1.7 + 1.3 + 2.8 + 2.8 + 1.6}{5} = 2.04, \qquad m_2 = \frac{1.6 + 1.3 + 1.4 + 2.6 + 2.7}{5} = 1.92. \tag{4.3}$$

In the assignment step, each measurement is reassigned to cluster with the closest mean,

1.7 *1.6* *1.3* *1.3* **2.8** *1.4* **2.8** **2.6** *1.6* **2.7.**

In the next update step, the two mean values can be computed as

$$m_1 = \frac{2.8 + 2.8 + 2.6 + 2.7}{4} = 2.725, \qquad m_2 = \frac{1.7 + 1.6 + 1.3 + 1.3 + 1.4 + 1.6}{6} = 1.483. \tag{4.4}$$

In the next assignment step, each measurement is again reassigned to cluster with the closest mean,

1.7 *1.6* *1.3* *1.3* **2.8** *1.4* **2.8** **2.6** *1.6* **2.7**.

Since this assignment is identical to the previous assignment, the algorithm stops. The output of the algorithm is a vector of cluster assignments, which in this example should be

2 2 2 2 1 2 1 1 2 1

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

`import numpy as np`
`print(clusterAnalysis(np.array([1.7, 1.6, 1.3, 1.3, 2.8, 1.4,`
`2.8, 2.6, 1.6, 2.7])))`
	`[2 2 2 2 1 2 1 1 2 1]`

Hand in on CodeJudge

This challenge can be be handed in on CodeJudge.

4G

Computer programs

Aims and objectives

After working through this exercise you should be able to:

- Create *computer programs* which:
  - Combines a script and multiple functions.
  - Interacts with the user by reading user-input from the keyboard and printing output to the screen.
- Use a loop to error-check user input.
- Structure a complex program by putting independent functionality into separate functions.
- Document your code:
  - Write clear and useful *comments* in your code.
  - Use comments to document user-defined functions.

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 4.1 + 4.4–4.10.

Video: Creating an interactive menu

Writing useful comments

Comments are important but often neglected. This typically happens because you are short of time. Neglecting to write comments can lead to problems if others are to read and understand your code, or if you have a long period of absence from the code yourself. Some rules of thumb that you should try to follow when writing comments:

- Write a minimal but sufficient amount of comments. Code without comments is generally bad, but too many comments is almost as bad, because excessive comments reduce code readability.
- Write the comments above the line of code it refers to. Comments should generally be short. A single comment should generally never be more than 1-2 lines long.
- The comments should clarify your intention with the code—not explain what the code does, which should be obvious from reading the code itself.
- A long piece of code should be partitioned into blocks where each block consists of one or more statements. Provide a short comment for each block as it makes it easier to understand the flow of the code.
- Particularly tricky statements should have a dedicated comment (i) when the purpose of the code is not obvious, or (ii) when it implements particularly complex functionality.
- When you declare a function, use comments to describe:
  - The purpose of the function and how it should be used.
  - The input arguments which it requires and how they should be used.
  - The output the function returns.
  - When and by who the function was created and/or modified.

Assignment 5A Temperature conversion

Temperature can be measured in degrees Fahrenheit, degrees Celsius, or Kelvin. We can convert between different units of temperature according to the following equations

	Celsius	Fahrenheit	Kelvin
Celsius	—	$F = 1.8 \cdot C + 32$	$K = C + 273.15$
Fahrenheit	$C = \frac{F - 32}{1.8}$	—	$K = \frac{F + 459.67}{1.8}$
Kelvin	$C = K - 273.15$	$F = 1.8 \cdot K - 459.67$	—

where C is the temperature in degrees Celsius, F is the temperature in degrees Fahrenheit, and K is the temperature in Kelvin.

Problem definition

Implement a function that converts temperatures between the three different units of measurement. The function must take three inputs: A decimal number `T`, which specifies the temperature and two strings, `unitFrom` and `unitTo` which specify the units to convert from and to. The strings must take one of the values `Celsius`, `Fahrenheit`, or `Kelvin`.

Solution template

```
def convertTemperature(T, unitFrom, unitTo):
  # Insert your code here
  return T
```

Input	
`T`	The input temperature (decimal number)
`unitFrom`	The unit of temperature of the input (string, `Celsius`, `Fahrenheit`, or `Kelvin`)
`unitTo`	The unit of temperature of the output (string, `Celsius`, `Fahrenheit`, or `Kelvin`)
Output	
`T`	The converted temperature (decimal number)

Example

If the input temperature `T` is 50 and `unitFrom` is the string `Fahrenheit` and `unitTo` is the string `Celsius`, we need to convert 50 degrees Fahrenheit to degrees Celsius. The converted temperature can be computed as follows:

$$C = \frac{F - 32}{1.8} = \frac{50 - 32}{1.8} = 10. \tag{5.1}$$

Thus, the output temperature `T` should be set to 10.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(convertTemperature(50.0, "Fahrenheit", "Celsius"))`	`10.0`

Discussion and futher analysis

Take some time to reflect on and discuss the way that you have commented your code. Do you follow all the suggestions given in section 5.2?

Input from the user

Until now, we have written the values of every variable directly into our scripts. If we wanted computations done for different values, we simply changed the values in the scripts. As we have written the scripts and functions ourselves, it is fairly easy to know exactly what to change.

Often, however, we want our code to be useful for users not familiar with the implementation details. To accomplish this, we need a way to receive values as inputs from the user, while the program is running. The `input` function will pause the program until the user has typed something on the keyboard and pressed enter. (Note: In Pyton 2.x the function is called `raw_input`). Whatever the user typed will then be returned by the function and can be stored in a variable. The program then continues and can use the variable in the further computations.

In order to prompt the user what to input, the function can be given a string as argument. The following code will prompt the user to enter a string (her name) which will be stored in the variable `name`

```
name = input("Please enter your name: ")
```

The following will prompt the user to enter a number (the lenght in meters). The string which the user types is converted from a string to a number using the function `float` and stored in the variable `len`

```
len = float(input("Enter the length in meters: "))
```

If you use this approach, you should be prepared to handle the error that might occur if the user does not type in a number. The following code shows one way to handle this. It will display an error message if the user does not input a number, and then prompt the user to try again. It uses a while-loop to repeatedly prompt the user until she types in a valid number, at which point it uses a break-statement to break out of the while-loop.

```
while True:
  try:
    len = float(input("Enter the length in meters: "))
    break
  except ValueError:
    print("Not valid number. Please try again.")
```

Look through the above code line by line, to see if you understand how it works. Especially, you should understand how the code uses the `try:` and `except ValueError:` construction to check whether the string which the user typed was correctly converted to a valid number. Once you understand the code, it will be easy to modify it: Maybe you would like to make further checks, for example to ensure that the number is within some specific range.

Handling numeric input in a function

If you need to get numeric input from the user multiple times in a computer program, it is a good idea to write a function that does the job for you. It can be as simple as the following, which simply loops until the user has typed in a valid number.

```
def inputNumber(prompt):
    # INPUTNUMBER Prompts user to input a number
    #
    # Usage: num = inputNumber(prompt) Displays prompt and asks user to input a
    # number. Repeats until user inputs a valid number.
    #
    # Author: Mikkel N. Schmidt, mnsc@dtu.dk, 2015

    while True:
        try:
            num = float(input(prompt))
            break
        except ValueError:
            pass

    return num
```

Exercise 5B Interactive temperature calculator

Write a script, that interacts with the user to convert a temperature from one unit to another. Make use of the function you implemented in assignment 5A. The program flow must be as follows:

1. The user is prompted to input a temperature (a decimal number).
2. The user is prompted to input the unit of the temperature (a string).
3. The user is prompted to input the unit of the temperature (a string) to convert to.
4. The program displays the converted temperature.

A session where the user interacts with the program might look as follows. The input given by the user is highlighted with red (the text should not be colored in your program.)

```
Please input a temperature: 50
Please input the unit of temperature (Celsius, Fahrenheit, or Kelvin): Fahrenheit
Please input the unit to convert to (Celsius, Fahrenheit, or Kelvin): Celsius
50 Fahrenheit = 10 Celsius
```

Discussion and futher analysis

How could you modify the code so that it:

- Is insensitive to the case of the input, i.e., allows the user to type for example `fahrenheit`, `FAHRENHEIT`, or `faHrenHeiT`?
- Also allows the user to just type `C`, `F`, or `K`?
- Gives suitable error messages and prompts the user to try again, if she does not input a number for the temperature or a correct string for the units?

5B

Exercise 5C A simple menu

In interactive computer programs it is sometimes useful to ask the user to choose between different options by displaying a simple menu. The following function, which uses the function `inputNumber` discussed previously, displays a menu and returns the number of the chosen menu item.

```
import numpy as np
from inputNumber import inputNumber

def displayMenu(options):
    # DISPLAYMENU Displays a menu of options, ask the user to choose an item
    # and returns the number of the menu item chosen.
    #
    # Usage: choice = displayMenu(options)
    #
    # Input    options   Menu options (array of strings)
    # Output   choice    Chosen option (integer)
    #
    # Author: Mikkel N. Schmidt, mnsc@dtu.dk, 2015

    # Display menu options
    for i in range(len(options)):
        print("{:d}. {:s}".format(i+1, options[i]))

    # Get a valid menu choice
    choice = 0
    while not(np.any(choice == np.arange(len(options))+1)):
        choice = inputNumber("Please choose a menu item: ")

    return choice
```

Examine the function to figure out how it works, and try it out (you can simply copy-paste it into Python). Remember that it uses the `inputNumber` function discussed previously, so you need to have that in place also.

Discussion and futher analysis

How could you modify the code so that it:

- Numbers the menu items beginning with zero rather than one?
- Numbers the menu items using letters rather than numbers?

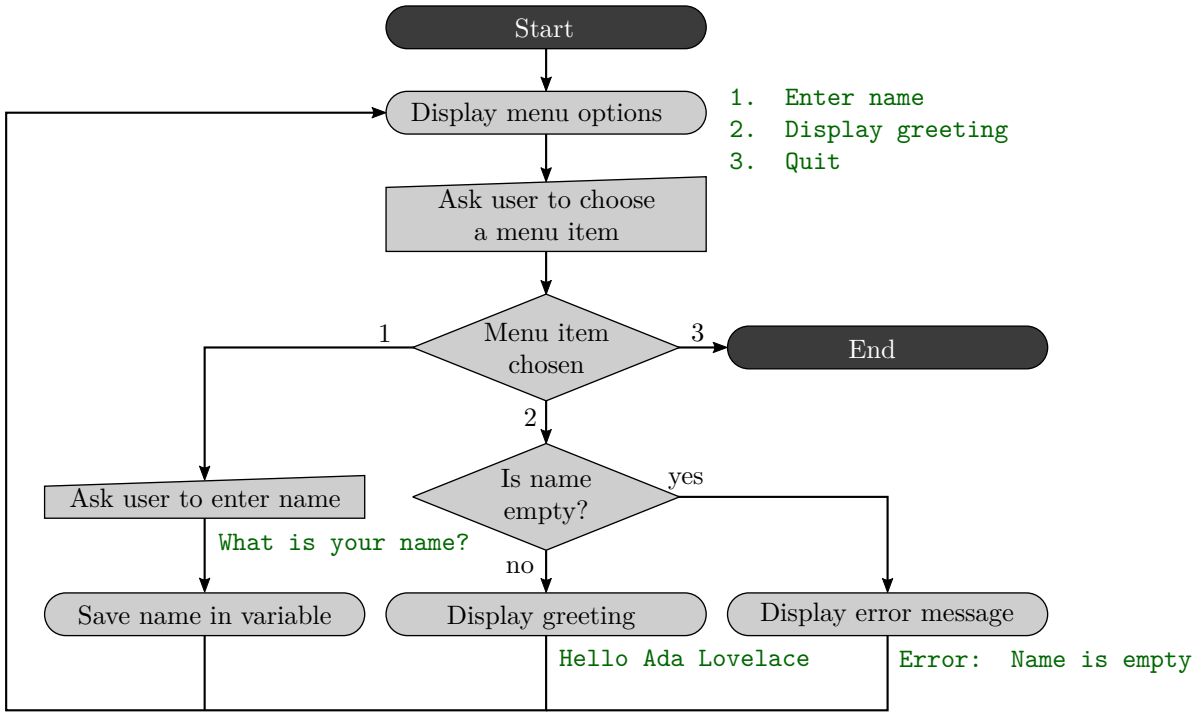
5C

Creating an interactive program main script

Let us consider a simple interactive program that does the following: When the program is run, it displays a menu with three options:

1. Enter name
2. Display greeting
3. Quit

If the user presses 1, she is asked to enter her name. After entering her name, the three menu options are shown again, and the program continues. If she presses 2, a greeting (including her name) is displayed. In the case that she has not yet entered her name, an error message is displayed instead. After the greeting or error message is displayed, the menu options are shown again, and the program continues. Finally, if the uses presses 3, the program stops. The program is illustrated in the following flowchart.



This interactive program can be implemented in many different ways: Since the program is supposed to run until the user chooses to quit, a common approach is to use an infinite loop to first display the menu options and then, inside the loop, use an `if`-statement to carry out the different tasks depending on the chosen menu option. If the option to quit the program is chosen, a `break` statement is used to break out of the infinite loop. A simple minimal implementation of the program is given in the following script.

- Read the script and see if you can understand how it works.
- Try to match up the flowchart with the script. Make sure you understand how the infinite `while`-loop and the `if`-statements inside the loop work.
- Run the program (copy-paste it into Python) and try it out.

```
# Import libraries
import numpy as np
from displayMenu import *

# Define menu items
menuItems = np.array(["Enter name", "Display greeting", "Quit"])

# Define empty name variable
name = ""

# Start
while True:
    # Display menu options and ask user to choose a menu item
    choice = displayMenu(menuItems)

    # Menu item chosen
    # ----------------------------------------------------------------
    # 1. Enter name
    if choice == 1:
        # Ask user to input name and save it in variable
        name = input("Please enter your name: ")

    # ----------------------------------------------------------------
    # 2. Display greeting
    elif choice == 2:
        # Is name empty?
        if name == "":
            # Display error message
            print("Error: Name is empty")
        else:
            # Display greeting
            print("Hello {:s}".format(name))

    # ----------------------------------------------------------------
    # 3. Quit
    elif choice == 3:
        # End
        break
```

Computer simulation with random numbers

In many situations it is very expensive or time-consuming to perform physical experiments. Instead, computer simulations can be used to approximate the results of real experiments. A simple way of setting up a simulation is as follows:

1. Define the possible inputs to the system.
2. Generate the inputs randomly from a probability distribution.
3. Perform some computation on the input, producing an output.

Once the simulation is set up, it is easy to run it a large number of times and aggregate the outputs. Such simulations are often referred to as *Monte Carlo* experiments.

As you have already seen, Python has functions for generating random numbers. For example, to create a random vector of length 4 you can use the function `np.random.rand` as follows

```
>> np.random.rand(4)
array([ 0.45522641,  0.36673742,  0.11908726,  0.25607802])
```

If you run the code multiple times you will get different numbers as output.

More accurately we refer to such generated numbers as *pseudo-random*, since they are in fact not random but a deterministic sequence of number that only appear to be random. Although this does not reflect the unpredictable randomness of real random events, the sequence is generated such that it matches the *statistical* behaviour of real random numbers.

The random number generator can be initialized to a *seed state*, such that will always generate the same sequence of "random" numbers. The seed, which is a non-negative integer, can be set using the `np.random.seed` function as follows,

```
>> np.random.seed(0)
>> np.random.rand(4)
array([ 0.5488135 ,  0.71518937,  0.60276338,  0.54488318])
```

If you run this code multiple times (each time resetting the seed to zero) you will get the same numbers as output.

When writing code which relies on pseudo-random numbers, it is often very useful to know the random seed (and thus the sequence of random numbers) that is used. Otherwise, testing and debugging the code or trying to reproduce results of a simulation will be difficult. It is therefore best if individual functions do not themselves use the random number generator in an undefined manner, but rather rely on recieving the random numbers they need as inputs when the function is called.

The random numbers generated above are all *uniformly distributed* between zero and one. To get a random number within an interval, say between ℓ and u, you can simply make the transformation

$$r^* = (u - \ell) \cdot r + \ell. \tag{5.2}$$

If r is a random number between 0 and 1, r^* will be a be between ℓ and u.

Python also has a function which directly generates uniform random numbers on an interval. For example, to generate four random numbers on the interval $[8, 9]$, you can write

```
>> np.random.uniform(8, 9, 4)
array([ 8.94737059,  8.73085581,  8.25394164,  8.21331198])
```

Exercise 5D Random numbers

We want to generate uniformly distributed random numbers, where the mean of the distribution given by μ and the width of the distribution is given by R. Thus, the uniform random number must be on the interval $\left[\mu - \frac{R}{2}, \mu + \frac{R}{2}\right]$. Although the mean of the distribution is equal to μ, if you generate, say, ten random numbers from this distribution it is not guaranteed that the *empirical* mean (the average) of the generated numbers will be equal to μ. In this exercise, we will investigate this futher.

Problem definition

Implement a function `randomSequence` that returns a sequence of N random numbers on the range described above, where μ, R and N are given as parameters to the function.

Write a script that does the following:

1. Use the function `randomSequence` to generate a vector of random numbers.
2. Calculate the empirical mean of the returned vector.

Solution template

```
def randomSequence(mu, R, N):
  # Insert your code here
  return r
```

Input	
`mu`	Mean of distribution (scalar decimal number)
`R`	Range of distribution (scalar decimal number)
`N`	Number of random draws (integer)
Output	
`r`	Random numbers (vector of decimal numbers)

Example

With $\mu = 5$, $R = 3$ and $N = 6$ the function will return a vector of 6 numbers in the interval $[3.5, 6.5]$. The function could for example return the numbers

$$4.1270, 3.9134, 5.6324, 6.0975, 3.2785, 5.5469 \tag{5.3}$$

The mean value of the random numbers in the returned list can be found as

$$\frac{4.1270 + 3.9134 + 5.6324 + 6.0975 + 3.2785 + 5.5469}{6} = 4.7660 \tag{5.4}$$

which is slightly lower than the mean value used to generate the numbers.

Discussion and futher analysis

Keep the mean and range parameters fixed and discuss the following:

- How does the empirical mean relate to the theoretical mean for different values of N (like 10, 100, 1000)?
- How does the empirical means compare for multiple runs with the same N?
- How could you empirically estimate the range?

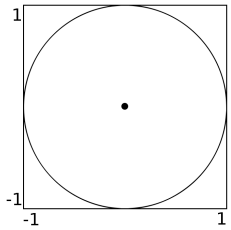
5D

Assignment 5E Monte Carlo estimation

The Monte Carlo method is a technique that can be used to approximate the distribution of random outcomes. This is done by recording the outcomes of running the same simulation multiple times. In this way, the method resembles repeated playing the same gambling game at a casino.

The Monte Carlo method is very useful, especially to approximate solutions to problems that are analytically difficult. In this assignment, we will use the Monte Carlo method in a simple scenario, to numerically estimate the area of a shape.

Consider a circle with radius 1, centered on $(0, 0)$.



By generating N random points uniformly on the the circumscribing square and counting how many points n that fall within the circle, we can estimate the area of the circle by multiplying the area of the square by the fraction of points inside the circle

$$A_{\text{circle}} \approx A_{\text{square}} \cdot \frac{n}{N}, \tag{5.5}$$

where in this case $A_{\text{square}} = 4$. To test if a point (x, y) is inside the circle, we can simply check if it magnitude of the vector from the center of the circle to (x, y) is less than one.

Problem definition

Write a function that estimates the area of a circle by Monte Carlo simulation. As input the function must receive two vectors containing the coordinates of the randomly drawn points. The function must return the estimated value of the area as output.

To test your function, you can use the function implemented in exercise 5D to generate a vector for the random x-values and a vector for the random y-values.

Solution template

```
def circleAreaMC(xvals, yvals):
  return A
```

Input	
`xvals`	The x-coordinates of points (list of decimal numbers)
`yvals`	The y-coordinates of points (list of decimal numbers)
Output	
`A`	Estimated value for the area of the circle (scalar decimal number)

Example

If we have randomly have drawn the following $N = 5$ points

$$(-0.1, 0.3)\ ,\ (0.7, -0.1)\ ,\ (0.8, 0.9)\ ,\ (0.5, 0.6)\ ,\ (-0.4, -0.3) \tag{5.6}$$

four of the points lies within the circle, and the area would be estimated as $A \approx 4 \cdot \frac{4}{5} = 3.2$.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`import numpy as np` `print(circleAreaMC(np.array([-0.1, 0.7, 0.8, 0.5, -0.4]), np.array([0.3, -0.1, 0.9, 0.6, -0.3])))`	`3.2`

Hand in on CodeJudge

The assignment must be handed in on CodeJudge.

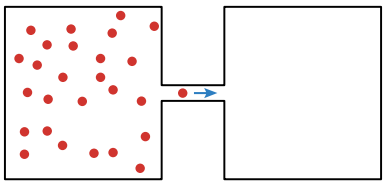
Discussion and futher analysis

- Try calling your function for different number of points (like 10, 1 000, and 1 000 000).
- Try calling your function multiple times with the same number of points.
- Try running the following code to plot an image of a circle along with your points.

```
import matplotlib.pyplot as plt
import numpy as np
p = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.sin(p), np.cos(p))
plt.plot(xvals, yvals, 'o')
plt.show()
```

Optional challenge 5F Thermodynamic simulation

Consider a physical experiment with two connected closed chambers of equal volume. The right chamber is initially empty, while the left contains N identical gas particles. We open a small hole between the chambers, such that the gas particles can move between the two chambers.



After some time, we expect that the system will reach an equilibrium, where the number of gas particles in the left chamber N_L is the same as the number of gas particles in the right chamber N_R. In this challenge we will use the Monte Carlo method to simulate the behaviour of the gas as it approaches this equilibrium.

The evolution of the system is considered as a series of time-steps, beginning at $t = 1$. At each time-step exactly one particle will pass through the hole, and we assume that the particles do not interact. The probability that a particle will move from the left to the right chamber is $p_{LR} = N_L/N$, and the probability of a particle will move from the right to the left chamber is $p_{RL} = 1 - p_{LR} = (N - N_L)/N$.

The simulation will iteratively proceed as follows:

1. Get a random number r from the interval $0 \leq r \leq 1$.
2. If $r \leq p_{LR}$, move one particle from the left to the right chamber. Otherwise move one particle from the right to the left chamber.
3. Repeat step 1 and 2 until $N_L = N_R$. Report back, how many time-steps it took to reach this equilibrium.

Problem definition

Write a function that performs the Monte Carlo simulation as described in the above algorithm. As input to the function, you have the number of particles (which must be an even number in order for an equilibrium to exist) and a vector of N random numbers between 0 and 1. The function must use the first random number in the vector as r at time $t = 1$, the second random number at time $t = 2$ etc. If the function runs out of random numbers before an equilibrium is reached, it must return the value $t = 0$ to indicate this.

Solution template

```
def thermoEquilibrium(N, r):
  # Insert your code here
  return t
```

Input	
`N`	Initial number of gas particles in the left chamber (even integer number)
`r`	Sequence of random numbers (vector of decimal numbers between 0 and 1)
Output	
`t`	Time-steps used to reach equilibrium (integer number) Return zero if equilibrium is not reached.

Example

Consider the simple case where there are only $N = 2$ particles. At $t = 1$ a particle will move from left to right (i.e., with probability $p_{LR} = \frac{2}{2} = 1$). When one particle moves to the right chamber, there will be exactly one particle in each chamber, and equilibrium has been reached. The number of time-steps to reach equilibrium is thus 1.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

```
import numpy as np
print(thermoEquilibrium(2.0, np.array([0.16, 0.04, 0.72,
      0.09, 0.17, 0.60, 0.26, 0.65, 0.69, 0.74, 0.45, 0.61,
      0.23, 0.37, 0.15, 0.83, 0.61, 1.00, 0.08, 0.44])))
```

```
1
```

Hand in on CodeJudge

This challenge can be handed in on CodeJudge.

Discussion and futher analysis

Test your function for different numbers of N (remember that N must be even in order for a equilibrium state to exist.)

- Does the number of time steps to reach equilibrium depend on the number of gas particles?
- Experiment with a large numbers of particles. Will the system always reach equilibrium?

5F

Files and matrices

Aims and objectives

After working through this exercise you should be able to:

- Use basic file system commands from within the programming language:
  - Show and list the contents of the current working directory.
  - Change the working directory.
- Read and write data files, including:
  - Data variables in Python's npz format.
  - Numeric data or text in plain text files.
  - Tabular data (numbers and text) in comma-separated-values (CSV) file.
- Initialize a matrix and perform basic matrix operations:
  - Indexing to extract or modify elements, rows, columns, and sub-matrices.
  - Determine the dimensions of the matrix.
  - Elementwise operations such as addition and multiplication.
  - Matrix operations including matrix-vector and matrix-matrix products.
  - Operations that work on matrices, such as computing the sum, mean, maximum, and minimum of each row, each column, or of all elements.
  - Concatenate matrices row- and columns-wise.
- Describe how to read data from at least one of the following formats using an external library or toolbox:
  - Spreadsheets in Microsoft Excel (xls) format.
  - Audio in Waveform Audio (wav) format.
  - Images in Joint Photographics Expert Group (jpeg) or Portable Network Graphics (png) file format.
  - Structured data in JavaScript Object Notation (json) or Extensible Markup Language (xml) format.

Suggested preparation

Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 8 + 14.1–14.5.

Files and directories

The *current working directory* is the directory in which Python will look for files, if not explicitly given a full path. When you create a program which loads or saves data in a data file, it is important to know what the current working directory is and how to change it.

To display the current working directory, you can type

```
>> import os
>> os.getcwd()
'/Users/alice'
```

If your are working in a project called MyProject stored in your documents folder it will probably be in a path similar to `/home/alice/MyProject` if you are using GNU/Linux, `/Users/alice/Documents/MyProject` if you are using OS X, or `C://Users/alice/My Documents/MyProject` if you are using Windows. To change the working directory to this folder, run the command

```
>>> os.chdir('/Users/alice/Documents/MyProject')
>>> os.getcwd()
'/Users/alice/Documents/MyProject'
```

To list the content of the current working directory type

```
>>> os.listdir(os.getcwd())
['data', 'myproject.txt']
```

As you can see, in this example there are two files in the working dir: One is a subdirectory named `data` and the other is a text-file named `myproject.txt`.

Loading and saving variables in files

There are several ways to save variables in files. One widely used is using the NumPy function `save` to store a single variable in a so-called npy-files or `savez` to store multiple variables in an npz-file. You can either use the absolute path or a path relative to the current working directory when loading or saving files.

The following example will create two variables, `x` and `y` and save them in a file `saved.npz`. Then it will delete the variables from the workspace. Finally, it will load the variables back into Python and display the contents of `x`.

```
>>> x = np.arange(1,10)
>>> y = np.sin(x)
>>> np.savez("saved.npz", x=x, y=y)
>>> del x
>>> del y
>>> tmp = np.load("saved.npz")
>>> x = tmp["x"]
>>> y = tmp["y"]
>>> x
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Hint

When you load the file `saved.npz` you store a NpzFile object in the variable `tmp`. This object has a `files` variable that contains a list of the variables stored in the npz-file. If you didn't save the files with the `x=x` and `y=y` the variables would have been named `arr_0` and `arr_1`.

```
>>> np.savez("saved.npz", x=x, y=y)
>>> tmp = np.load("saved.npz")
>>> tmp.files
['y', 'x']
>>> np.savez("saved.npz", x, y)
>>> tmp = np.load("saved.npz")
>>> tmp.files
['arr_1', 'arr_0']
```

It is good practice to name your variables when saving them in an npz-file.

Exercise 6A Loading and saving variables

For this exercise download the file `smooth.npz`. This file contains two variables named `x` and `y`. Do the following:

1. Change the current working directory to the place where you have saved the file.
2. List the content of the working directory to verify that the file is there.
3. Load the file into Python such that the two variables are called `x` and `y`.
4. Plot the content of the variable `x` and `y`. It should look roughly like an inverted U-shape.

> **Hint**
>
> You can plot the variables using the command `plt.plot(x, y)`. Remember to first import the matplotlib.pyplot module using `import matplotlib.pyplot as plt` and to call `plt.show()` to display the plot..

5. Change `y` such that it contains 1 minus the original value of `y`.
6. Save `x` and the new `y` in a new file.
7. Restart your Python editor and make sure that you can then load and plot the data in the new file.

6A

Working with matrices

A matrix is a collection of data (numbers) organized in a two-dimensional grid. For example the following matrix, A, contains the numbers from 1 to 6 organized on a 2-by-3 grid:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}. \tag{6.1}$$

The matrix above is said to have two rows and three columns.

Creating matrices To create a matrix in Python, you can use the `np.array` from the NumPy library to convert a list (in square brackets) where each element is a list (again in square brackets) with the numbers that go into each row of the matrix. For example, the matrix A above can be created as follows:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
```

Converting between vectors and matrices It is sometimes useful to convert between a vector and a matrix. A vector can be converted into a matrix using the `np.reshape`-function, and a matrix can be converted into a vector by reshaping it with the dimension set to `-1`.

```
>>> a = np.arange(1, 7)
>>> a
array([1, 2, 3, 4, 5, 6])
>>> A = np.reshape(a, [2, 3])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> b = np.reshape(A, -1)
>>> b
array([1, 2, 3, 4, 5, 6])
```

> Hint
>
> Note, that when converting between matrices and vectors in this way, Python operates on the matrix in a row-by-row manner. If you wish to work column-by-column you may need to transpose the matrix.

Accessing elements Accessing elements in a matrix works in much the same way as for a vector. The elements in a matrix can be accessed using square brackets. For example to access the element in the second row at the third column, you may write

```
>>> A[1, 2]
6
```

You can also use this notation to change an element in a matrix.

A whole row or column can be accessed in a similar way, by replacing the row or column index by a colon.

```
>>> A[0, :]
array([1, 2, 3])
>>> A[:, 1]
array([2, 5])
```

You can use the colon operator as an index to extract a sub-matrix. For example, to get the sub-matrix consisting of all rows for columns 2 to 3, you may write:

```
>>> A[:, 1:3]
array([[2, 3],
       [5, 6]])
```

Concatenating matrices You can concatenate two matrices with compatible dimensionality using the `vstack` and `hstack` functions.

If we define the following matrices

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}, \quad C = \begin{bmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \\ 17 & 18 & 19 \end{bmatrix}, \tag{6.2}$$

we may write

```
>>> AB = np.hstack((A, B))
>>> AB
array([[ 1,  2,  3,  7,  8],
       [ 4,  5,  6,  9, 10]])
>>> AC = np.vstack((A, C))
>>> AC
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

Matrix operations You can use all the same element-wise operations on matrices that you have used on vectors. In addition you might be interested in computing other properties of a matrix, such as the row- or columns-wise minimum, maximum, sum, and average. To do this, you can use the functions `np.amin`, `np.amax`, `np.sum`, and `np.mean`, and specify whether the function should be applied for each row or column as demonstrated below. Furthermore, the `shape` function will tell you the dimensions of the matrix.

```
>>> np.amin(A, axis=0)
array([1, 2, 3])
>>> np.amin(A, axis=1)
array([1, 4])
>>> np.sum(A, axis=0)
array([5, 7, 9])
>>> np.sum(A, axis=1)
array([ 6, 15])
>>> A.shape
(2, 3)
```

Matrix multiplication Matrix-matrix and matrix-vector multiplication can be performed using the `dot` function. For example, the matrix vector product

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 28 \end{bmatrix}, \tag{6.3}$$

can be computed as

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = np.array([3, 2, 1])
>>> np.dot(A, b)
array([10, 28])
```

Matrix transposition You can transpose a matrix using the `t` function as follows:

```
>>> A.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Assignment 6B Production cost

You want to compare the cost of producing some different items. To produce each item, a number of different resources are required. For each item you know how many units of each type of resource is required. This information is given as a matrix, where each column corresponds to an item and each row corresponds to a resource. The values in the matrix denote how many units of the resource is required for the item. Furhtermore, you know the cost of each resource, which is given as a vector.

Problem definition

Create a function that computes the cost of each item and returns it as a vector. Find an elegant and simple way to write the code using what you have learned about matrix and vector operations.

Solution template

```
def computeItemCost(resourceItemMatrix, resourceCost):
  # Insert your code here
  return itemCost
```

Input	
`resourceItemMatrix`	A matrix describing how many units of each resource is required for each item.
`resourceCost`	Vector of the cost of each resource.
Output	
`itemCost`	Vector of the cost of each item.

Example

Say that the number of resources required for each item is given by the following matrix,

$$\begin{array}{r} \\ \text{Resource 1:} \\ \text{Resource 2:} \\ \text{Resource 3:} \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 6 & 3 & 0 \\ 17 & 11 & 9 \\ 4 & 2 & 12 \end{bmatrix} \end{array}, \tag{6.4}$$

which shows that it requires 6 units of resource 1, 17 units of resource 2, and 4 units of resource 3 to produce one item A, and similar for item B and item C.

Further, let us assume that the cost of each resource is given by the vector,

$$\begin{bmatrix} 101.25, & 84.00, & 75.50 \end{bmatrix}, \tag{6.5}$$

The cost of producing item A is then

$$6 \cdot 101.25 + 17 \cdot 84.00 + 4 \cdot 75.5 = 2337.5 \tag{6.6}$$

Similarly the cost of item B and item C can be computed as 1378.75 and 1662.00. Thus, the final result should be the vector

$$[2337.50, \quad 1378.50, \quad 1662.00] \tag{6.7}$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

`import numpy as np`
`print(computeItemCost(np.array([[6,3,0],[17,11,9],[4,2,12]]),`
`  np.array([101.25,84.00,75.50])))`
	`[ 2337.5  1378.75  1662.  ]`

Hand in on CodeJudge

This exercise must be handed on CodeJudge.

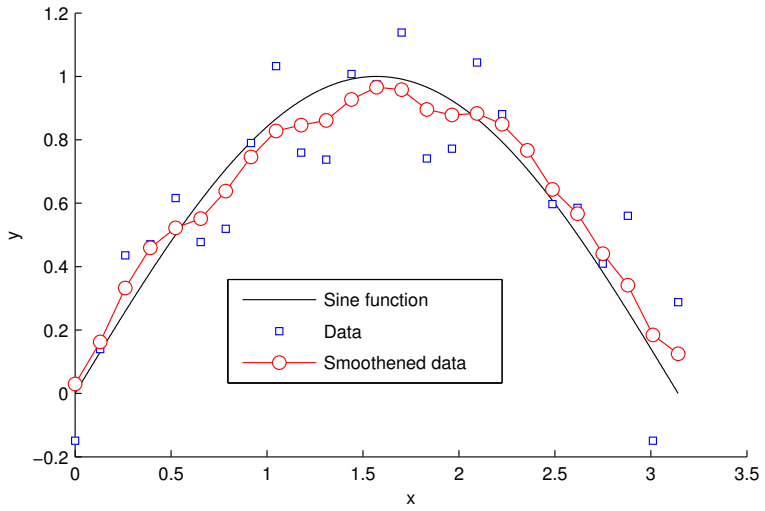
6B

Assignment 6C Moving average

When recording any type of noisy data, that be audio signals or stock prices, noise is often reduced by *smoothening* the data. One algorithm of smoothing a signal is the n-sample symmetric weighted moving average. Let us say we are given a signal as a vector y with values y_i for $i \in (1, \ldots, N)$. Then, a five-sample symmetric weighted moving average, $\hat{y}$, can be computed by

$$\hat{y}_i = \frac{y_{i-2} + 2 \cdot y_{i-1} + 3 \cdot y_i + 2 \cdot y_{i+1} + y_{i+2}}{9}, \tag{6.8}$$

i.e., as a weighted average of the surrounding datapoints. To compute the first two and last two smoothened signal values, we must make some assumption about the signal outside its bounds: We will assume that y_i is zero for $i < 1$ and $i > N$. The algorithm is illustrated below. The figure shows a noisy measurement of a sine wave function and a five-sample symmetric weighted moving average.



Problem definition

Create a function that takes a signal vector as input and computes the five-sample weighted moving average of the signal.

Solution template

```
def movingAvg(y):
  # Insert your code here
  return ysmooth
```

Input	
`y`	Input signal (vector)
Output	
`ysmooth`	Five-sample moving average smoothing of input signal (vector)

Example

If your signal file consists of a vector

$$y = [0.8, 0.9, 0.7, 0.6, 0.3, 0.4] \tag{6.9}$$

We can solve the smoothing problem by making use of a matrix: We first construct a matrix where each row is

a shifted and scaled version of the signal as illustrated below:

$$\begin{array}{cccc} 1 & \cdot & 0.8 & 0.9 \\ 2 & \cdot & & 0.8 \\ 3 & \cdot & & \\ 2 & \cdot & & \\ 1 & \cdot & & \end{array} \begin{bmatrix} 0.7 & 0.6 & 0.3 & 0.4 & 0 & 0 \\ 0.9 & 0.7 & 0.6 & 0.3 & 0.4 & 0 \\ 0.8 & 0.9 & 0.7 & 0.6 & 0.3 & 0.4 \\ 0 & 0.8 & 0.9 & 0.7 & 0.6 & 0.3 \\ 0 & 0 & 0.8 & 0.9 & 0.7 & 0.6 \end{bmatrix} \begin{array}{cc} & \\ & \\ & \\ 0.4 & \\ 0.3 & 0.4 \end{array} = \begin{bmatrix} 0.7 & 0.6 & 0.3 & 0.4 & 0 & 0 \\ 1.8 & 1.4 & 1.2 & 0.6 & 0.8 & 0 \\ 2.4 & 2.7 & 2.1 & 1.8 & 0.9 & 1.2 \\ 0 & 1.6 & 1.8 & 1.4 & 1.2 & 0.6 \\ 0 & 0 & 0.8 & 0.9 & 0.7 & 0.6 \end{bmatrix}. \quad (6.10)$$

In the first row, y is shifted left twice; in the second row y is shifted left once and multiplied by two; in the third row y is multiplied by three; etc. Summing each column and dividing by 9 yields the final result

$$\hat{y} = [0.54444, 0.7, 0.68889, 0.56667, 0.4, 0.26667]. \quad (6.11)$$

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

```
import numpy as np
print(movingAvg(np.array([0.8, 0.9, 0.7, 0.6, 0.3, 0.4])))
```

```
[ 0.54444444  0.7         0.68888889  0.56666667  0.4
  0.26666667]
```

Hand in on CodeJudge

In order to validate you can either upload a file containing your implementation of the function or directly copy the code into codejudge.

Discussion and futher analysis

Try loading the variables `x` and `y` from the data file `smooth.npz` which contains the noisy sine function data used to generate the figure above. You should then be able to reproduce the plot in the figure using the following code:

```
import matplotlib.pyplot as plt
plt.plot(x, np.sin(x), "k")
plt.plot(x, y, "g.")
plt.plot(x, movingAvg(y), "r.-")
plt.show()
```

6C

Reading text files

There are several ways to read text from a plain text-file. One is to use the `readlines` function. The following example will read all the characters of the file `small_text.txt` into the variable `smalltxt`.

```
>>> filein = open("small_text.txt", "r") # Opens the file for reading
>>> lines = filein.readlines()           # Reads all lines into an array
>>> smalltxt = "".join(lines)            # Joins the lines into one big string
```

This will load all the text in the file into a single string variable.

Assignment 6D Frequency of letters

A simple way to analyze a text document is to examine how often the different letters in the alphabet occur in the text. As you will see in a later assignment, this can be used for example to detect the language in which the text is written. In this assignment we will create a function that takes the filename of a text file as input and computes the frequency of the letters in the text. In order to simplify the problem we will convert the text to lower case and only consider the characters from `a` to `z`.

First, we must read in the text file and convert it to lower case. Then we should count the number of times each of the letters `a` to `z` occur. Here it might be a good idea to create a string or vector with all the letters in the alphabet: Using that, we can make a loop over the letters in order to count how many times they each occur in the text. Finally, based on the occurence counts we can compute the frequency.

Problem definition

Create a function that takes a text file as input and returns a vector of size 26 with the frequency in percent of each character `a`, `b`, ... `z` (not sensitive to case.) The frequency must be computed as the number of occurences divided by the total number of characters from `a` to `z` that occur, multiplied by one hundred. All other letters such as `ø` and `ä` as well as all symbols must be ignored.

Solution template

```
def letterFrequency(filename):
  # Insert your code here
  return freq
```

Input	
`filename`	A string that is the filename of a plain text file in the current working directory.
Output	
`freq`	A vector of length 26 containing the frequency of occurrence of each of the 26 letters from a to z.

Example

In the file `small_text.txt` is a small example. Counting the number of occurences of each of the letters yields the result shown below (second row). Dividing each number by the total number of occurences and multiplying with 100 yields the frequency (last row).

Letter	`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`	`i`	`j`	`k`	`l`	`m`
Occurrences	105	29	32	59	160	26	25	87	93	1	15	44	14
Frequency (%)	8.10	2.24	2.47	4.55	12.35	2.01	1.93	6.71	7.18	0.08	1.16	3.40	1.08

Letter	`n`	`o`	`p`	`q`	`r`	`s`	`t`	`u`	`v`	`w`	`x`	`y`	`z`
Occurrences	87	102	19	1	78	70	142	37	12	38	0	20	0
Frequency (%)	6.71	7.87	1.47	0.08	6.02	5.40	10.96	2.85	0.93	2.93	0.00	1.54	0.00

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

`import numpy as np`
`print(letterFrequency("small_text.txt"))`
	`[ 8.10185185 2.23765432 2.4691358 4.55246914 12.34567901`
	`2.00617284 1.92901235 6.71296296 7.17592593 0.07716049`
	`1.15740741 3.39506173 1.08024691 6.71296296 7.87037037`
	`1.46604938 0.07716049 6.01851852 5.40123457 10.95679012`
	`2.85493827 0.92592593 2.93209877 0. 1.54320988 0. ]`

Hand in on CodeJudge

This exercise must be handed on CodeJudge.

6D

Working with CSV files

Comma-separated values (CSV) or character-separated values is a fileformat that stores data (numbers and text) in a plain-text form. The values in a CSV file are delimited by some separator character that is often a comma or a semicolon. It is a widely used data format, which is used by many different programs.

There are several possibilites when working with CSV files in Python. We will use a DataFrame object from the Pandas module to store the content of a CSV file. This is a flexible data type that you can use in a way that resembles the way you use a spreadsheet or an Excel file.

> Hint
>
> The DataFrame object is part of the Pandas module. To import Pandas write `import pandas as pd` similar to how you have imported the Numpy module.

For the following examples we will use the `topscorers_small.csv` file that contains the 5 topscorers from the 2013–2014 season of the Champions League. You can open the file in a text editor to see that it simply 6 lines of values (strings and numbers) separated by commas. The first line in the file contains the titles of the data columns in the file. Make sure you have the file in the current working directory when trying out the following.

To load a CSV file into a DataFrame use the `pd.read_csv` function.

```
>>> topscorers = pd.read_csv("topscorers_small.csv")
>>> topscorers
               Player                  Team  Goals  MinutesPlayed
0   Cristiano Ronaldo           Real Madrid     17            993
1  Zlatan Ibrahimovic   Paris Saint-Germain     10            670
2         Diego Costa       Atletico Madrid      8            580
3        Lionel Messi             Barcelona      8            630
4       Sergio Aguero       Manchester City      6            429
```

To retrieve the names of the columns write `topscorers.columns` to get the columns attribute.

```
>>> topscorers.columns
Index([u'Player', u'Team', u'Goals', u'MinutesPlayed'], dtype='object')
>>> topscorers.columns[1]
'Team'
```

It is easy to use the names to directly index that column of the DataFrame.

```
>>> topscorers.Goals
0    17
1    10
2     8
3     8
4     6
Name: Goals, dtype: int64
>>> np.array(topscorers.Goals)
array([17, 10,  8,  8,  6])
```

Hint

When working with DataFrame objects functions such as `topscorers.columns` or `topscorers.Goals` does not return a numpy array. However it is possible to convert a DataFrame object or part of a DataFrame object to a NumPy array. For example, the entire DataFrame can easily be converted into an array:

```
>>> np.array(topscorers)
array([['Cristiano Ronaldo', ' Real Madrid', 17, 993],
       ['Zlatan Ibrahimovic', ' Paris Saint-Germain', 10, 670],
       ['Diego Costa', ' Atletico Madrid', 8, 580],
       ['Lionel Messi', ' Barcelona', 8, 630],
       ['Sergio Aguero', ' Manchester City', 6, 429]], dtype=object)
```

That way you can get a vector that can be used for the usual kinds of computations. If for instance you want the number of minutes played per goal scored you can type

```
>>> np.array(topscorers.MinutesPlayed) / np.array(topscorers.Goals)
array([ 58.41176471,  67.        ,  72.5       ,  78.75      ,  71.5       ])
```

To get the names of the players, you can write

```
>>> topscorers.Player
0     Cristiano Ronaldo
1    Zlatan Ibrahimovic
2           Diego Costa
3          Lionel Messi
4         Sergio Aguero
Name: Player, dtype: object
```

It is also possible to access individual elements of the DataFrame.

```
>>> topscorers.Goals[3]
8
>>> topscorers.Player[3]
'Lionel Messi'
```

You can also retrieve columns or rows of the DataFrame without using the column names directly. Instead use the `iloc` member and then index the DataFrame as you would a NumPy array. Note that it is not directly a NumPy array that is returned. In order to get a NumPy array, use the `np.array` function as usual.

```
>>> topscorers.iloc[:,3]
0    993
1    670
2    580
3    630
4    429
Name: MinutesPlayed, dtype: int64
>>> topscorers.iloc[2,:]
Player               Diego Costa
Team              Atletico Madrid
Goals                          8
MinutesPlayed                580
Name: 2, dtype: object
>>> np.array(topscorers.iloc[:,3])
array([993, 670, 580, 630, 429])
>>> np.array(topscorers.iloc[2,:])
array(['Diego Costa', ' Atletico Madrid', 8, 580], dtype=object)
```

This is, for instance, useful when you want to perform some operation on each of the columns in the DataFrame.

Assignment 6E Language detection

Different languages use the different letters with different frequencies and this can be used to determine in which language a text is written. In this exercise you should use the function created in the previous exercise to compute the frequencies of letters in a text. Given this vector of frequencies you can compute the *squared error* between the frequencies in the text and the (average) frequencies in the language.

$$E_\ell = \sum_{i=1}^{26} (F_i^t - F_i^\ell)^2 \tag{6.12}$$

where F_i^t is the frequency of letter i in the text and F_i^ℓ is the frequency of letter i in the language. The language which has the lowest squared error is the one that best matches the text in terms of the letter frequency.

The frequencies of the letters in fifteen different languages are given in the file `letter_frequencies.csv`.[1] A snapshot of a part the file is given below.

Letter	English	French	German	Spanish	Portuguese	Esperanto	Italian	Turkish
a	8.167	7.636	6.516	11.525	14.634	12.117	11.745	12.92
b	1.492	0.901	1.886	2.215	1.043	0.98	0.927	2.844
c	2.782	3.26	2.732	4.019	3.882	0.776	4.501	1.463
d	4.253	3.669	5.076	5.51	4.992	3.044	3.736	5.206
e	12.702	14.715	16.396	12.681	11.57	8.995	11.792	9.912
f	2.228	1.066	1.656	0.692	1.023	1.037	1.153	0.461
g	2.015	0.866	3.009	1.768	1.303	1.171	1.644	1.253
h	6.094	0.737	4.577	0.703	0.781	0.384	0.636	1.212
i	6.966	7.529	6.55	6.247	6.186	10.012	10.143	9.6
j	0.153	0.613	0.268	0.443	0.397	3.501	0.011	0.034

Problem definition

Create a function that takes as input a vector of frequencies of occurrences of letters in a text. The function must read the file `letter_frequencies.csv`, compute the squared error for each language, and return a vector of squared errors for the fifteen languages.

Solution template

```
def computeLanguageError(freq):
  # Insert your code here
  return E
```

Input	
`freq`	A vector of size 26 containing the frequency of the letters `a`–`z` in a text.
Output	
`se`	A vector of length 15 containing the squared error between the input vector and each of the 15 languages in the CSV file

Example

Let the vector in the following table be the input vector and the CSV file as the `letter_frequencies.csv` from CampusNet. This should give the following squared error for the first 10 languages:

English	French	German	Spanish	Portuguese	Esperanto	Italian	Turkish	Swedish	Polish
9.04	108.24	99.55	121.02	165.54	164.75	128.56	211.07	89.98	190.64

Example test case

[1] Source: http://en.wikipedia.org/wiki/Letter_frequency

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output

```
import numpy as np
print(computeLanguageError(np.array([8.101852, 2.237654,
      2.469136, 4.552469, 12.345679, 2.006173, 1.929012,
      6.712963, 7.175926, 0.077160, 1.157407, 3.395062,
      1.080247, 6.712963, 7.870370, 1.466049, 0.077160,
      6.018519, 5.401235, 10.956790, 2.854938, 0.925926,
      2.932099, 0.000000, 1.543210, 0.000000])))
```

```
[   9.03927     108.23630662   99.54527245  121.0194921
     165.54454939
  164.74825044  128.56084094  211.07248403   89.98061244
      190.64402388
   93.79889711  112.93292492  192.24702032  173.1080387
       134.53866161]
```

Hand in on CodeJudge

This exercise must be handed on CodeJudge.

Discussion and futher analysis

This method of only using the letter frequencies to determine a language is not very efficient, specially when the text of which you are trying to identify the language of is only a few words.

The method can be extended to use bigrams or trigrams giving a quite robust method of identifying the language of a text that is actually used in practice. Bigrams are two adjacent characters in a word instead of single characters. The two most frequent bigrams in english are ‘th’ and ‘he’.

6E

Exercise 6F Advanced file types

Choose one of the filetypes

1. Spreadsheets in Microsoft Excel (xls) format.
2. Audio in Waveform Audio (wav) format.
3. Images in Joint Photographics Expert Group (jpeg) or Portable Network Graphics (png) file format.
4. Structured data in JavaScript Object Notation (json) or Extensible Markup Language (xml) format.

Download the file of the appropriate type from CampusNet. Look through the documentation or search online for a function or external library that can read the file.

Read the file into Python such that you can work with the data in the file.

6F

Plotting and strings

Aims and objectives

After working through this exercise you should be able to:

- Visualize numeric data by generating graphical plots including:
  - Scatter plots.
  - Line graphs.
  - Histograms.
- Create plots that visualize multiple data series.
- Add a title, axis labels, grid lines, and a data legend to a plot.
- Customize properties of the graph including:
  - Axis limits.
  - Line styles, markers, and colors.
- Manipulate strings
  - Concatenate strings and extract substrings
  - Find, replace, and separate strings

Suggested preparation
Pyplot tutorial: http://matplotlib.org/users/pyplot_tutorial.html
Downey, "Think Python: How to Think Like a Computer Scientist", Chapter 2.8.

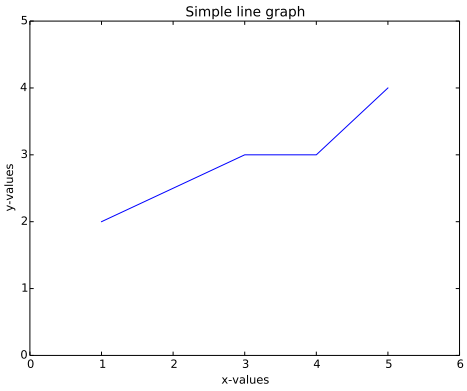
Working with plots

Data acquisition and data analysis play important roles in engineering and science. Typically the result of an experiment is a set of data: A possibly large set of numbers that need interpretation. It is often desirable to visialize data sets in the form of graphical plots. Visualization often makes it easier for you to get a good sense of how the data behaves, and to discover patterns and trends in the data. A visual representation of data also makes it easier for you to communicate your results to an audience.

Line graphs

Creating graphical plots in Python is quite easy. Try copying the following code into a script and running it, to make a line graph of four data points.

```
import numpy as np                  # Import NumPy
import matplotlib.pyplot as plt     # Import the matplotlib.pyplot module
x = (1, 3, 4, 5)                    # Make some data, x- and y-values
y = (2, 3, 3, 4)
plt.plot(x, y)                      # Plot line graph of x and y
plt.title("Simple line graph")      # Set the title of the graph
plt.xlabel("x-values")              # Set the x-axis label
plt.ylabel("y-values")              # Set the y-axis label
plt.xlim([0, 6])                    # Set the limits of the x-axis
plt.ylim([0, 5])                    # Set the limits of the y-axis
plt.show()
```



Closely examining the code above will give you a good idea of how to make graphical plots: Make sure that you understand each step above. The first two arguments of the `plt.plot` command are the x- and y-values of the points that will be drawn. The `plt.xlabel` and `plt.ylabel` commands specify the axis labels, and the `plt.title` command sets the title of the plot. The `plt.xlim` and `plt.ylim` commands specify the upper and lower limits of the x- and y-axis.

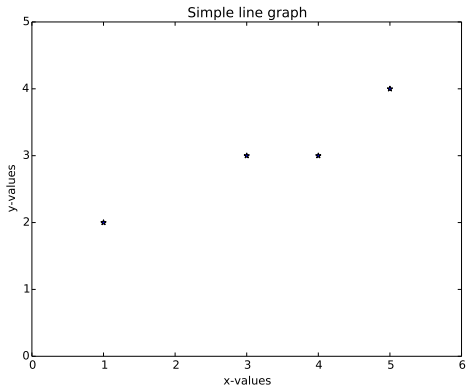
We use the `matplotlib` module to create plots, and it must therefore be loaded together with the `numpy` module. All plotting functions will be applied to the same figure, until the `show` command is called, after which Python will begin on a new blank figure.

Scatter plots

In addition to line graphs, you can also use a similar approach to make scatter plots, in which each data point (x- and y-coordinate) is displayed as a symbol without connecting lines.

```
import numpy as np                  # Import NumPy
import matplotlib.pyplot as plt     # Import the matplotlib.pyplot module
x = (1, 3, 4, 5)                    # Make some data, x- and y-values
y = (2, 3, 3, 4)
plt.plot(x, y, "b*")                # Scatter plot with blue stars
```

```
plt.title("Simple scatter plot")    # Set the title of the graph
plt.xlabel("x-values")              # Set the x-axis label
plt.ylabel("y-values")              # Set the y-axis label
plt.xlim([0, 6])                    # Set the limits of the x-axis
plt.ylim([0, 5])                    # Set the limits of the y-axis
plt.show()
```

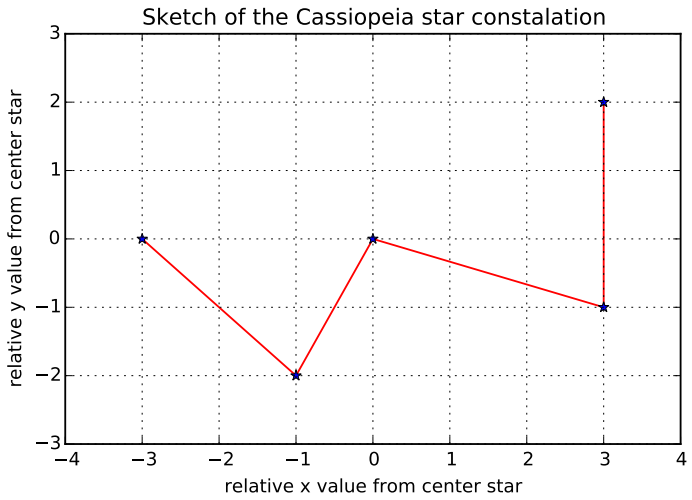


The third argument `"b*"` specify how the points should be drawn—in this case as blue stars. Type `help(plt.plot)` to read about the different markers and colors that can be used. It is also possible to draw point markers and connecting lines at the same time using a single command—see how in the documentation.

Exercise 7A Cassiopeia graph

Replicate the following figure, where the points are shown as blue stars connected by red solid lines. Make sure the points are connected in the same way as in the plot, that the axes have the correct range, and that the title and labels are correct. As seen below, you must also add a grid in the background, which makes it easier to read off the values of the points.

> **Hint**
>
> A grid is added by calling `plt.grid()`.



Exercise 7B Scatter plot

Create two different vectors, x and y, each containing 2000 random numbers, uniformly distributed between -10 and 10.

Make a scatter plot of the points x, y for which the following two conditions are met:

$$\max(|x|, |y|) > 5 \quad \text{and} \quad \sqrt{x^2 + y^2} < 10 \tag{7.1}$$

The points that do not satisfy these conditions must be ignored (not plotted). The plotted points must be drawn with blue circles. The result should look like a fuzzy circle with a square hole.

7B

Exercise 7C Histograms

A histogram is a plot type used in statistics to show frequencies (how many times elements within given intervals appear in a vector). Each interval is shown as a bin, with the height representing the number of elements falling within the interval.

We will use a histogram to graphically show the distribution of heads and tails in a sequence of N independent fair coin tosses. For each toss the probability of seeing a head is equal to the probability of seeing a tail, $P_H = P_T = \frac{1}{2}$. The following code will create a random sequence x that simulates the results of 10 coin tosses. In x, seeing a head is represented by False while seeing a tail is represented by True.

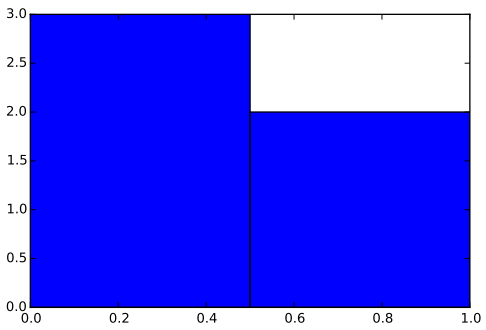
```
import numpy as np
x = np.random.rand(10) < 0.5
```

In the histogram, the x-axis must contain two bins. The height of the first bin represents the number of heads N_H seen in the sequence, while the height of the second represents the number of tails N_T.

This can be shown as a histogram with two bins by the command:

```
plt.hist(x, 2)
```

If $x = [0, 1, 0, 0, 1]$, we have seen 3 heads and 2 tails, resulting in the histogram showed in the following figure:



Problem definition

1. Simulate sequences of fair coin tosses and plot histograms for $N = 10$, $N = 100$ and $N = 1000$. Remember to set title and labels for your histogram appropriately.
   - From which of the histograms can you best see that $P_H = P_T = \frac{1}{2}$?
   - How would you change the code to generate a sequence of tosses with an unfair coin, where $P_H = 0.7$ and $P_T = 0.3$?
2. Write a script that simulates a sequence of N throws of a fair six-sided die and plots the distribution of the outcomes in a histogram with 6 bins. Run your code for $N = 10$, $N = 100$ and $N = 1000$ throws.

   > **Hint**
   >
   > You can get the sequence of outcomes (with even probabilities) using the command `np.ceil(6 * np.random.rand(N))`.

   - How large must N be, before the histogram clearly shows that the probability of each outcome is identical (equal to $\frac{1}{6}$).
   - Compared to the coin-toss example, why do we in general need a longer sequence of dice throws to approximate the underlying probability distribution in the histogram?

Exercise 7D Radiocarbon dating

Carbon is a naturally occurring mineral in living organisms. A small part of the carbon will be of the radioactive isotope carbon 14. While and organism is alive, the carbon within its tissue will continuously be replenished from the various nutrients it ingests. When the organism dies, the carbon content is no longer replenished, and the carbon 14 isotopes within the tissue will slowly decay, such that only the stable carbon 12 remains. The half-life of carbon 14 is approximately $t_{1/2} = 5700$ years, meaning that every 5700 years the amount of carbon 14 in the dead organic material will be halved.

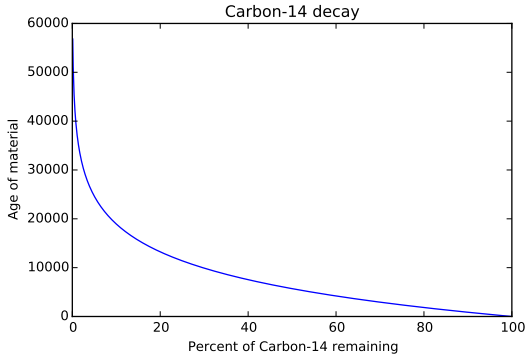
Archaeologists use measurements of carbon 14 to estimate how long time ago an organism died. This can be estimated as follows,

$$t = -\frac{1}{\lambda} \cdot \ln\left(\frac{N}{N_0}\right), \tag{7.2}$$

where t is the estimated time in years since the organism died, N is the amount of remaining carbon 14, N_0 is the amount of carbon 14 when the organism was alive (this can for instance be estimated from knowing the content of living samples of the same type of organism) and λ is the decay rate which can be computed from the half-life as $\lambda = \frac{\ln(2)}{t_{1/2}}$.

Problem definition

Create a plot that shows the exponential decay of carbon 14 in organic material. The plot must show the age of the organic material on the y-axis and the corresponding percentage of remaining carbon 14 on the x-axis (such that $N_0 = 100$ and $0 \leq N \leq 100$). Use a lineplot to illustrate this by creating a list of x-values and a list of the corresponding y-values. The more points you use, the smoother the curve will look. With 1000 points, you will get a plot as shown in the following figure. Give your plot a meaningful title and labels for the axis.



- A tusk of a woolly mammoth discovered in Siberia is analysed. The ratio between carbon 12 and 14 is found to only be one tenth of the ratio measured from ivory of recently deceased (present day) elephants. Look at your plot to verify that the mammoth is around 19,000 years old.
- Carbon 14 measurements are considered unreliable for estimating organic material older than 50,000 years. Look at your plot and consider why this is so.

Exercise 7E Temperature in the UK

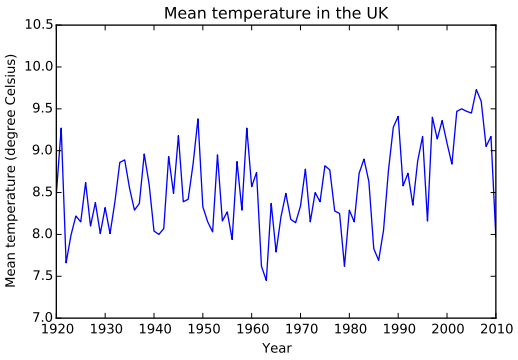
Download the UKTemperature.csv dataset. This dataset contains yearly temperature information for the United Kingdom, for the years 1912 to 2012 in comma-separated values (CSV) format.[1] Each row contains temperature data for a particular year. For each year there are 14 values:

Column	Attribute
1	Year
2–13	Jan, Feb, etc.: Mean temperature for each month
14	Average: Yearly mean temperature

All temperatures are measured in degrees Celsius.

Problem definition

Make a line graph with the year on the x-axis and the yearly mean temperature on the y-axis. Set the range of the x-axis to begin at 1920 and end at 2010. Your plot should look like the one in the following figure. Make sure to set appropriate labels and title.



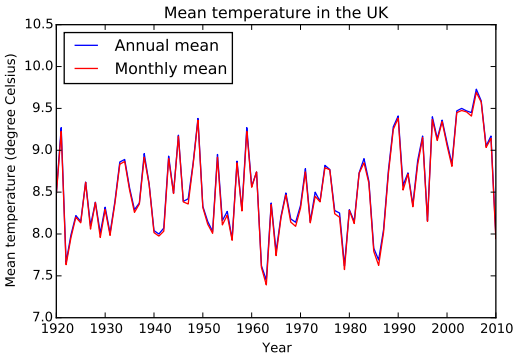
For each year, calculate the average of the mean temperatures of the 12 months and show this as another line in the same plot, but in a different color. Make sure to use appropriate labels and title.

Add a legend to the plot, to inform the reader about the difference between the two timeseries. In order to show legends in a plot, each timeseries must be given a label (name), when it is plotted. The `legend` command can then be used to show legends with these names. Try to run the following code to see how it works, and read the documentation to learn more about making legend, for example to control where the legend is placed.

```
x = np.arange(1,20)
y1 = x
y2 = x**2
plt.plot(x, y1, label="Series 1")
plt.plot(x, y2, label="Series 2")
plt.legend(loc="upper left")
```

Your plot should look like the following:

[1] Based on data from the UK's National Weather Service http://www.metoffice.gov.uk



- Why is the mean temperature for the entire year not exactly equal to the 12 months average?
- Does it make sense that the annual mean in general seems to be higher than the mean of the 12 months?

7E

Exercise 7F Saving and printing plots

Continue working with a plot you have made in one of the previous exercises. Save the plot in a file without using the graphical user interface.

> Hint
>
> You can use the `plt.savefig` function to save a plot into a file. Use the documentation to see how this works.

Once you have saved the plot in a file, make sure that you can insert it in your favorite word processing system, such as LaTeX or Microsoft Word.

7F

Working with strings

You have already used strings for many different purposes, such as getting input into your program from the user or from a file, and outputting information to the user either on the screen or by saving it in a file. The following is a short introduction to working with strings in Python where we will go through some of the most important basic string operations.

Concatenating strings

To concatenate means to put together, one after another. To concatenate strings (without any separating characters inbetween) you ca use the plus operator.

```
>>> str1 = "This"
>>> str2 = "is"
>>> str3 = "easy"
>>> str1 + str2 + str3
'Thisiseasy'
```

If you would like to concatenate the strings by separating them with a character such as a space or a dash, you can use the the `join` method.

```
>>> " ".join((str1, str2, str3))
'This is easy'
>>> "-".join((str1, str2, str3))
'This-is-easy'
```

Extracting substrings

To extract a substring or perhaps a single character from a string, you can index the string just like a vector.

```
>>> str = "This-is-easy"
>>> str[5]
'i'
>>> str[5:10]
'is-ea'
```

Find and replace in strings

If you want to find out if/where a substring occurs inside a longer string, you can use the `find` method.

```
>>> str = "This-is-easy"
>>> str.find("is")
2
```

The `find` method returns the index of the first match. If you want to find all the substring matches, you can for example use a loop repeatedly search for the next match until you have found all the matches.

```
>>> index = 0
>>> while True:
...     index = str.find("is", index)
...     if index==-1:
...         break
...     print(index);
...     index += 1
2
5
```

If the substring does not match anywhere `find` returns -1.

```
>>> str.find("not there")
-1
```

To replace all occurences of a substring within a string you can use the `replace` method.

```
>>> str.replace("is", "eta")
'Theta-eta-easy'
```

Splitting strings into tokens

Sometimes you need to split a string into substrings separated by some character (such as space, comma, dash etc.) You could of course use the methods we just described to find and extract the substrings, but since this is a somewhat common task there is also a special function called `split` that is much easier to use:

```
>>> str = "This-is-easy"
>>> tokens = str.split("-")
>>> tokens
['This', 'is', 'easy']
>>> tokens[1]
'is'
```

Vectors of strings

If you need to handle multiple strings, you can create a list or "vector" of strings and work with it in a way similar to how you work with vectors of numbers.

```
>>> my_strings = np.array(["This is one string", "Here is another", "This is the third"])
>>> my_strings
array(['This is one string', 'Here is another', 'This is the third'],
      dtype='<U18')
>>> my_strings[2]
'This is the third'
```

Assignment 7G The NATO alphabet

The NATO alphabet is useful for spelling words over a noisy telephone connection.

A Alpha	B Bravo	C Charlie	D Delta	E Echo	F Foxtrot	G Golf	H Hotel	I India
J Juliet	K Kilo	L Lima	M Mike	N November	O Oscar	P Papa	Q Quebec	R Romeo
S Sierra	T Tango	U Uniform	V Victor	W Whiskey	X Xray	Y Yankee	Z Zulu	

Problem definition

Create a function that spells out a word (written in plain text) in the NATO alphabet. The function should accept input in upper, lower, or mixed case, and should output the NATO "code words" exactly as written above, separated by dashes. You can assume that the input string contains only letters from a-z (A-Z).

Solution template

```
def textToNato(plainText):
  # Insert your code here
  return natoText
```

Input	
`plainText`	Input word in plain text (string)
Output	
`natoText`	Output in dash-separated NATO alphabet (string)

Example

For example, given the input `hello`, the function should return the string `Hotel-Echo-Lima-Lima-Oscar`.

Example test case

Remember to thoroughly test your code before handing it in. You can test your solution on the example above by running the following test code and checking that the output is as expected.

Test code	Expected output
`print(textToNato("hello"))`	`Hotel-Echo-Lima-Lima-Oscar`

Hand in on CodeJudge

In order to validate you can either upload a file containing your implementation of the function or directly copy the code into codejudge.

Discussion and futher analysis

How could you create a function that does the reverse, i.e., converts a word written in the NATO alphabet (as output by your function) back into plain text? If you are up for the task, create such a function and test thoroughly that it works correctly by converting words first to the NATO alphabet and then back and checking that the result is identical to the original input.