This page was generated from basics.ipynb.
Open in Colab

Exercise 1 (hello world)	Exercise 2 (compliment)	Exercise 3 (multiplication)
Exercise 4 (multiplication table)	Exercise 5 (two dice)	Exercise 6 (triple square)
Exercise 7 (areas of shapes)	Exercise 8 (solve quadratic)	Exercise 9 (merge)
Exercise 10 (detect ranges)	Exercise 11 (interleave)	Exercise 12 (distinct characters)
Exercise 13 (reverse dictionary)	Exercise 14 (find matching)	Exercise 15 (two dice comprehension)
Exercise 16 (transform)	Exercise 17 (positive list)	Exercise 18 (acronyms)
Exercise 19 (sum equation)	Exercise 20 (usemodule)	

Python

Basic concepts

Basic input and output

The traditional “Hello, world” program is very simple in Python. You can run the program by selecting the cell by mouse and pressing control-enter on keyboard. Try editing the string in the quotes and rerunning the program.

[1]:
```
print("Hello world2!")
```
```
Hello world2!
```

Multiple strings can be printed. By default, they are concatenated with a space:

[2]:
```
print("Hello,", "John!", "How are you?")
```
```
Hello, John! How are you?
```

In the print function, numerical expression are first evaluated and then automatically converted to strings. Subsequently the strings are concatenated with spaces:

[3]:
```
print(1, "plus", 2, "equals", 1+2)
```
```
1 plus 2 equals 3
```

Reading textual input from the user can be achieved with the input function. The input function is given a string parameter, which is printed and prompts the user to give input. In the example below, the string entered by the user is stored the variable `name`. Try executing the program in the interactive notebook by pressing control-enter!

[4]:
```
name=input("Give me your name: ")
print("Hello,", name)
```

```
Give me your name: Jarkko
Hello, Jarkko
```

Indentation

Repetition is possible with the for loop. Note that the body of for loop is indented with a tabulator or four spaces. Unlike in some other languages, braces are not needed to denote the body of the loop. When the indentation stops, the body of the loop ends.

[5]:
```
for i in range(3):
    print("Hello")
print("Bye!")
```

```
Hello
Hello
Hello
Bye!
```

Indentation applies to other compound statements as well, such as bodies of functions, different branches of an if statement, and while loops. We shall see examples of these later.

The `range(3)` expression above actually results with the sequence of integers 0, 1, and 2. So, the range is a half-open interval with the end point excluded from the range. In general, expression range(n) gives integers 0, 1, 2, ..., n-1. Modify the above program to make it also print the value of variable i at each iteration. Rerun the code with control-enter.

Exercise 1 (hello world)

Fill in the missing piece in the solution stub file `hello_world.py` in folder `src` to make it print the following:

```
Hello, world!
```

Make sure you use correct indenting. You can run it with command `python3 src/hello_world.py`. If the output looks good, then you can test it with command `tmc test`. If the tests pass, submit your solution to the server with command `tmc submit`.

Exercise 2 (compliment)

Fill in the stub solution to make the program work as follows. The program should ask the user for an input, and the print an answer as the examples below show.

```
What country are you from? Sweden
I have heard that Sweden is a beautiful country.

What country are you from? Chile
I have heard that Chile is a beautiful country.
```

/

Exercise 3 (multiplication)

Make a program that gives the following output. You should use a for loop in your solution.

```
4 multiplied by 0 is 0
4 multiplied by 1 is 4
4 multiplied by 2 is 8
4 multiplied by 3 is 12
4 multiplied by 4 is 16
4 multiplied by 5 is 20
4 multiplied by 6 is 24
4 multiplied by 7 is 28
4 multiplied by 8 is 32
4 multiplied by 9 is 36
4 multiplied by 10 is 40
```

Variables and data types

We saw already earlier that assigning a value to variable is very simple:

[6]:
```
a=1
print(a)
```

```
1
```

Note that we did not need to introduce the variable `a` in any way. No type was given for the variable. Python automatically detected that the type of `a` must be `int` (an integer). We can query the type of a variable with the builtin function `type`:

[7]:
```
type(a)
```

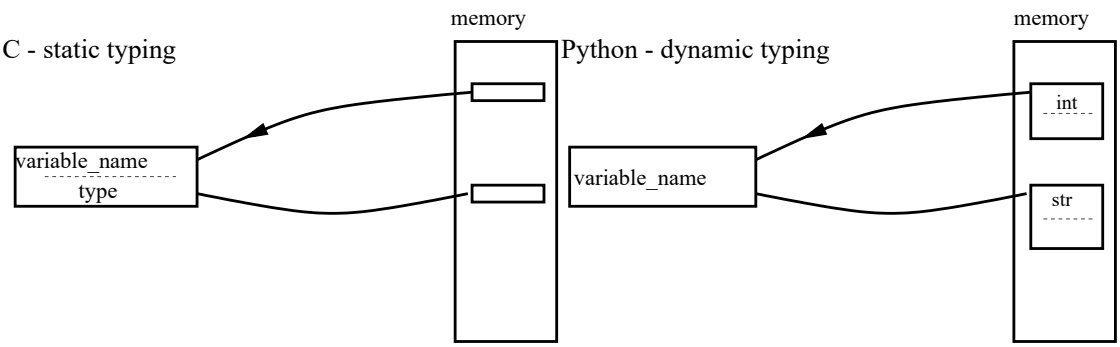
[7]:
```
int
```

Note also that the type of a variable is not fixed:

[8]:
```
a="some text"
type(a)
```

[8]:
```
str
```

In Python the type of a variable is not attached to the name of the variable, like in C for instance, but instead with the actual value. This is called dynamic typing.



We say that a variable is a name that *refers* to a value or an object, and the assignment operator *binds* a variable name to a value.

The basic data types in Python are: `int`, `float`, `complex`, `str` (a string), `bool` (a boolean with values `True` and `False`), and `bytes`. Below are few examples of their use.

[9]:
```
i=5
f=1.5
b = i==4
print("Result of the comparison:", b)
c=0+2j                                # Note that j denotes the imaginary unit of complex
numbers.
print("Complex multiplication:", c*c)
s="conca" + "tenation"
print(s)
```

```
Result of the comparison: False
Complex multiplication: (-4+0j)
concatenation
```

The names of the types act as conversion operators between types:

[10]:
```
print(int(-2.8))
print(float(2))
print(int("123"))
print(bool(-2), bool(0))  # Zero is interpreted as False
print(str(234))
```

```
-2
2.0
123
True False
234
```

A *byte* is a unit of information that can represent numbers between 0 and 255. A byte consists of 8 *bits*, which can in turn represent either 0 or 1. All the data that is stored on disks or transmitted across the internet are sequences of bytes. Normally we don't have to care about bytes, since our strings and other variables are automatically converted to a sequence of bytes when needed to. An example of the correspondence between the usual data types and bytes is the characters in a string. A single character is encoded as a sequence of one or more bytes. For example, in the common UTF-8 encoding the character `c` corresponds to the byte with integer value 99 and the character `ä` corresponds to sequence of bytes [195, 164]. An example conversion between characters and bytes:

[11]:
```
b="ä".encode("utf-8")     # Convert character(s) to a sequence of bytes
print(b)                  # Prints bytes in hexadecimal notation
print(list(b))            # Prints bytes in decimal notation
```

```
b'\xc3\xa4'
[195, 164]
```

[12]:
```
bytes.decode(b, "utf-8")  # convert sequence of bytes to character(s)
```

[12]:
```
'ä'
```

During this course we don't have to care much about bytes, but in some cases, when loading data sets, we might have to specify the encoding if it deviates from the default one.

Creating strings

A string is a sequence of characters commonly used to store input or output data in a program. The characters of a string are specified either between single (`'`) or double (`"`) quotes. This optionality is useful if, for example, a string needs to contain a quotation mark: “I don’t want to go!”. You can also achieve this by *escaping* the quotation mark with the backslash: ‘I don’t want to go’.

The string can also contain other escape sequences like `\n` for newline and `\t` for a tabulator. See literals for a list of all escape sequences.

[13]:
```
print("One\tTwo\nThree\tFour")
```

```
One     Two
Three   Four
```

A string containing newlines can be easily given within triple double or triple single quotes:

[14]:
```
s="""A string
spanning over
several lines"""
```

Although we can concatenate strings using the `+` operator, for effiency reasons, one should use the `join` method to concatenate larger number of strings:

[15]:
```
a="first"
b="second"
print(a+b)
print(" ".join([a, b, b, a]))   # More about the join method later
```

```
firstsecond
first second second first
```

Sometimes printing by concatenation from pieces can be clumsy:

[16]:
```
print(str(1) + " plus " + str(3) + " is equal to " + str(4))
# slightly better
print(1, "plus", 3, "is equal to", 4)
```

```
1 plus 3 is equal to 4
1 plus 3 is equal to 4
```

The multiple catenation and quotation characters break the flow of thought. *String interpolation* offers somewhat easier syntax.

There are multiple ways to do sting interpolation:

- Python format strings
- the `format` method
- f-strings

Examples of these can be seen below:

[3]:
```
print("%i plus %i is equal to %i" % (1, 3, 4))     # Format syntax

print("{} plus {} is equal to {}".format(1, 3, 4)) # Format method

print(f"{1} plus {3} is equal to {4}")             # f-string
```

```
1 plus 3 is equal to 4
1 plus 3 is equal to 4
1 plus 3 is equal to 4
```

/

The `i` format specifier in the format syntacs corresponds to integers and the specifier `f` corresponds to floats. When using f-strings or the `format` method, integers use `d` instead. In format strings specifiers can usually be omitted and are generally used only when specific formatting is required. For example in f-strings `f"{4:3d}"` would specify the number 4 left padded with spaces to 3 digits.

It is often useful to specify the number of decimals when printing floats:

[2]:
```
print("%.1f %.2f %.3f" % (1.6, 1.7, 1.8))              # Old style
print("{:.1f} {:.2f} {:.3f}".format(1.6, 1.7, 1.8))    # newer style
print(f"{1.6:.1f} {1.7:.2f} {1.8:.3f}")                # f-string
```

```
1.6 1.70 1.800
1.6 1.70 1.800
1.6 1.70 1.800
```

The specifier `s` is used for strings. An example:

[4]:
```
print("%s concatenated with %s produces %s" % ("water", "melon", "water"+"melon"))
print("{0} concatenated with {1} produces {0}{1}".format("water", "melon"))
print(f"{'water'} concatenated with {'melon'} produces {'water' + 'melon'}")
```

```
water concatenated with melon produces watermelon
water concatenated with melon produces watermelon
water concatenated with melon produces watermelon
```

Look here for more details about format specifiers, and for comparison between the old and new style of string interpolation.

Different ways of string interpolation have different strengths and weaknesses. Generally choosing which to use is a matter of personal preference. On this course examples and model solutions will predominantly use f-strings and the `format` method.

Expressions

An *expression* is a piece of Python code that results in a value. It consists of values combined together with *operators*. Values can be literals, such as `1`, `1.2`, `"text"`, or variables. Operators include arithmetics operators, comparison operators, function call, indexing, attribute references, among others. Below there are a few examples of expressions:

```
1+2
7/(2+0.1)
a
cos(0)
mylist[1]
c > 0 and c !=1
(1,2,3)
a<5
obj.attr
(-1)**2 == 1
```

Note that in Python the operator `//` performs integer division and operator `/` performs float division. The `**` operator denotes exponentiation. These operators might therefore behave differently than in many other common languages.

As another example the following expression computes the kinetic energy of a non-rotating object: `0.5 * mass * velocity**2`

Statements

Statements are commands that have some effect. For example, a function call (that is not part of another expression) is a statement. Also, the variable assignment is a statement:

[21]:
```
i = 5
i = i+1    # This is a commong idiom to increment the value of i by one
i += 1     # This is a short-hand for the above
```

Note that in Python there are no operators `++` or `--` unlike in some other languages.

It turns out that the operators `+ - * / // % & | ^ >> << **` have the corresponding *augmented assignment operators* `+= -= *= /= //= %= &= |= ^= >>= <<= **=`

Another large set of statements is the flow-control statements such as if-else, for and while loops. We will look into these in the next sections.

Loops for repetitive tasks

In Python we have two kinds of loops: `while` and `for`. We briefly saw the `for` loop earlier. Let's now look at the `while` loop. A `while` loop repeats a set of statements while a given condition holds. An example:

[22]:
```
i=1
while i*i < 1000:
    print("Square of", i, "is", i*i)
    i = i + 1
print("Finished printing all the squares below 1000.")
```

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
Square of 6 is 36
Square of 7 is 49
Square of 8 is 64
Square of 9 is 81
Square of 10 is 100
Square of 11 is 121
Square of 12 is 144
Square of 13 is 169
Square of 14 is 196
Square of 15 is 225
Square of 16 is 256
Square of 17 is 289
Square of 18 is 324
Square of 19 is 361
Square of 20 is 400
Square of 21 is 441
Square of 22 is 484
Square of 23 is 529
Square of 24 is 576
Square of 25 is 625
Square of 26 is 676
Square of 27 is 729
Square of 28 is 784
Square of 29 is 841
Square of 30 is 900
Square of 31 is 961
Finished printing all the squares below 1000.
```

Note again that the body of the while statement was marked with the indentation.

Another way of repeating statements is with the `for` statement. An example

[23]:
```
s=0
for i in [0,1,2,3,4,5,6,7,8,9]:
    s = s + i
print("The sum is", s)
```

```
The sum is 45
```

The `for` loop executes the statements in the block as many times as there are elements in the given list. At each iteration the variable `i` refers to another value from the list in order. Instead of the giving the list explicitly as above, we could have used the *generator* `range(10)` which returns values from the sequence 0,1,...,9 as the for loop asks for a new value. In the most general form the `for` loop goes through all the elements in an *iterable*. Besides lists and generators there are other iterables. We will talk about iterables and generators later this week.

When one wants to iterate through all the elements in an iterable, then the `for` loop is a natural choice. But sometimes `while` loops offer cleaner solution. For instance, if we want to go through all Fibonacci numbers up till a given limit, then it is easier to do with a `while` loop.

Exercise 4 (multiplication table)

In the `main` function print a multiplication table, which is shown below:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

For example at row 4 and column 9 we have 4*9=36.

Use two nested for loops to achive this. Note that you can use the following form to stop the `print` function from automatically starting a new line:

[24]:
```
print("text", end="")
print("more text")
```

```
textmore text
```

Print the numbers in a field with width four, so that the numbers are nicely aligned. For instructions on how adjust the field width refer to pyformat.info.

Decision making with the if statement

The if-else statement works as can be expected. Try running the below cell by pressing control+enter.

/

[25]:
```
x=input("Give an integer: ")
x=int(x)
if x >= 0:
    a=x
else:
    a=-x
print("The absolute value of %i is %i" % (x, a))
```

```
Give an integer: -1
The absolute value of -1 is 1
```

The general from of an if-else statement is

```
if condition1:
    statement1_1
    statement1_2
    ...
elif condition2:
    statement2_1
    statement2_2
    ...
...
else:
    statementn_1
    statementn_2
    ...
```

Another example:

[26]:
```
c=float(input("Give a number: "))
if c > 0:
    print("c is positive")
elif c<0:
    print("c is negative")
else:
    print("c is zero")
```

```
Give a number: 3
c is positive
```

Breaking and continuing loop

Breaking the loop, when the wanted element is found, with the `break` statement:

[27]:
```
l=[1,3,65,3,-1,56,-10]
for x in l:
    if x < 0:
        break
print("The first negative list element was", x)
```

```
The first negative list element was -1
```

Stopping current iteration and continuing to the next one with the `continue` statement:

[3]:
```
from math import sqrt, log
l=[1,3,65,3,-1,56,-10]
for x in l:
    if x < 0:
        continue
    print(f"Square root of {x} is {sqrt(x):.3f}")
    print(f"Natural logarithm of {x} is {log(x):.4f}")
```

```
Square root of 1 is 1.000
Natural logarithm of 1 is 0.0000
Square root of 3 is 1.732
Natural logarithm of 3 is 1.0986
Square root of 65 is 8.062
```

```
Natural logarithm of 65 is 4.1744
Square root of 3 is 1.732
Natural logarithm of 3 is 1.0986
Square root of 56 is 7.483
Natural logarithm of 56 is 4.0254
```

Exercise 5 (two dice)

Let us consider throwing two dice. (A dice can give a value between 1 and 6.) Use two nested `for` loops in the `main` function to iterate through all possible combinations the pair of dice can give. There are 36 possible combinations. Print all those combinations as (ordered) pairs that sum to 5. For example, your printout should include the pair `(2,3)`. Print one pair per line.

Functions

A function is defined with the `def` statement. Let's do a doubling function.

[29]:
```
def double(x):
    "This function multiplies its argument by two."
    return x*2
print(double(4), double(1.2), double("abc")) # It even happens to work for strings!
```

```
8 2.4 abcabc
```

The double function takes only one parameter. Notice the *docstring* on the second line. It documents the purpose and usage of the function. Let's try to access it.

[30]:
```
print("The docstring is:", double.__doc__)
help(double)   # Another way to access the docstring
```

```
The docstring is: This function multiplies its argument by two.
Help on function double in module __main__:

double(x)
    This function multiplies its argument by two.
```

Most of Python's builtin functions, classes, and modules should contain a docstring.

[31]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Here's another example function:

[32]:
```
def sum_of_squares(a, b):
    "Computes the sum of arguments squared"
    return a**2 + b**2
print(sum_of_squares(3, 4))
```

```
25
```

/

> Note the terminology: in the function definition the names a and b are called parameters of the function; in the function call, however, 3 and 4 are called arguments to the function.

It would be nice that the number of arguments could be arbitrary, not just two. We could pass a list to the function as a parameter.

[33]:
```
def sum_of_squares(lst):
    "Computes the sum of squares of elements in the list given as parameter"
    s=0
    for x in lst:
        s += x**2
    return s
print(sum_of_squares([-2]))
print(sum_of_squares([-2,4,5]))
```

```
4
45
```

This works perfectly! There is however some extra typing with the brackets around the lists. Let's see if we can do better:

[34]:
```
def sum_of_squares(*t):
    "Computes the sum of squares of arbitrary number of arguments"
    s=0
    for x in t:
        s += x**2
    return s
print(sum_of_squares(-2))
print(sum_of_squares(-2,4,5))
```

```
4
45
```

The strange looking argument notation (the star) is called *argument packing*. It packs all the given positional arguments into a tuple `t`. We will encounter tuples again later, but it suffices now to say that tuples are *immutable* lists. With the `for` loop we can iterate through all the elements in the tuple.

Conversely, there is also syntax for *argument unpacking*. It has confusingly exactly same notation as argument packing (star), but they are separated by the location where used. Packing happens in the parameter list of the functions definition, and unpacking happens where the function is called:

[35]:
```
lst=[1,5,8]
print("With list unpacked as arguments to the functions:", sum_of_squares(*lst))
# print(sum_of_squares(lst))    # Does not work correctly
```

```
With list unpacked as arguments to the functions: 90
```

The second call failed because the function tried to raise the list of numbers to the second power. Inside the function body we have `t=([1,5,8])`, where the parentheses denote a tuple with one element, a list.

In addition to positional arguments we have seen so far, a function call can also have *named arguments*. An example will explain this concept best:

[36]:
```
def named(a, b, c):
    print("First:", a, "Second:", b, "Third:", c)
named(5, c=7, b=8)
```

/

```
First: 5 Second: 8 Third: 7
```

Note that the named arguments didn't need to be in the same order as in the function definition. The named arguments must come after the positional arguments. For example, the following function call is illegal `named(a=5, 7, 8)`.

One can also specify an optional parameter by giving the parameter a default value. The parameters that have default values must come after those parameters that don't. We saw that the parameters of the `print` function were of form `print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`. There were four parameters with default values. If some default values don't suit us, we can give them in the function call using the name of the parameter:

[37]:
```
print(1, 2, 3, end=' |', sep=' -*- ')
print("first", "second", "third", end=' |', sep=' -*- ')
```

```
1 -*- 2 -*- 3 |first -*- second -*- third |
```

We did not need to specify all the parameters with default values, only those we wanted to change.

Let's go through another example of using parameters with default values:

[38]:
```
def length(*t, degree=2):
    """Computes the length of the vector given as parameter. By default, it computes
    the Euclidean distance (degree==2)"""
    s=0
    for x in t:
        s += abs(x)**degree
    return s**(1/degree)
print(length(-4,3))
print(length(-4,3, degree=3))
```

```
5.0
4.497941445275415
```

With the default parameter this is the Euclidean distance, and if $p \neq 2$ it is called p-norm.

We saw that it was possible to use packing and unpacking of arguments with the * notation, when one wants to specify arbitrary number of *positional arguments*. This is also possible for arbitrary number of named arguments with the `**` notation. We will talk about this more in the data structures section.

Visibility of variables

Function definition creates a new *namespace* (also called local scope). Variables created inside this scope are not available from outside the function definition. Also, the function parameters are only visible inside the function definition. Variables that are not defined inside any function are called `global variables`.

Global variable are readable also in local scopes, but an assignment creates a new local variable without rebinding the global variable. If we are inside a function, a local variable hides a global variable by the same name:

[39]:
```
i=2           # global variable
def f():
    i=3       # this creates a new variable, it does not rebind the global i
    print(i)  # This will print 3
```

```
f()
print(i)      # This will print 2
```

```
3
2
```

If you really need to rebind a global variable from a function, use the `global` statement. Example:

[40]:
```
i=2
def f():
    global i
    i=5       # rebind the global i variable
    print(i)  # This will print 5
f()
print(i)      # This will print 5
```

```
5
5
```

Unlike languages like C or C++, Python allows defining a function inside another function. This *nested* function will have nested scope:

[41]:
```
def f():            # outer function
    b=2
    def g():        # inner function
        #nonlocal b # Without this nonlocal statement,
        b=3         # this will create a new local variable
        print(b)
    g()
    print(b)
f()
```

```
3
2
```

Try first running the above cell and see the result. Then uncomment the nonlocal stamement and run the cell again. The `global` and `nonlocal` statements are similar. The first will force a variable refer to a global variable, and the second will force a variable to refer to the variable in the nearest outer scope (but not the global scope).

Exercise 6 (triple square)

Write two functions: `triple` and `square`. Function `triple` multiplies its parameter by three. Function `square` raises its parameter to the power of two. For example, we have equalities `triple(5)==15` and `square(5)==25`.

Part 1.

In the `main` function write a `for` loop that iterates through values 1 to 10, and for each value prints its triple and its square. The output should be as follows:

```
triple(1)==3 square(1)==1
triple(2)==6 square(2)==4
...
```

Part 2.

Now modify this `for` loop so that it stops iteration when the square of a value is larger than the triple of the value, without printing anything in the last iteration.

Note that the test cases check that both functions `triple` and `square` are called exactly once per iteration.

Exercise 7 (areas of shapes)

Create a program that can compute the areas of three shapes, triangles, rectangles and circles, when their dimensions are given.

An endless loop should ask for which shape you want the area be calculated. An empty string as input will exit the loop. If the user gives a string that is none of the given shapes, the message "unknown shape!" should be printed. Then it will ask for dimensions for that particular shape. When all the necessary dimensions are given, it prints the area, and starts the loop all over again. Use format specifier `f` for the area.

What happens if you give incorrect dimensions, like giving string "aa" as radius? You don't have to check for errors in the input.

Example interaction:

```
Choose a shape (triangle, rectangle, circle): triangle
Give base of the triangle: 20
Give height of the triangle: 5
The area is 50.000000
Choose a shape (triangle, rectangle, circle): rectangel
Unknown shape!
Choose a shape (triangle, rectangle, circle): rectangle
Give width of the rectangle: 20
Give height of the rectangle: 4
The area is 80.000000
Choose a shape (triangle, rectangle, circle): circle
Give radius of the circle: 10
The area is 314.159265
Choose a shape (triangle, rectangle, circle):
```

Data structures

The main data structures in Python are strings, lists, tuples, dictionaries, and sets. We saw some examples of lists, when we discussed `for` loops. And we saw briefly tuples when we introduced argument packing and unpacking. Let's get into more details now.

Sequences

A *list* contains arbitrary number of elements (even zero) that are stored in sequential order. The elements are separated by commas and written between brackets. The elements don't need to be of the same type. An example of a list with four values:

```
[42]: [2, 100, "hello", 1.0]
```

```
[42]: [2, 100, 'hello', 1.0]
```

A *tuple* is fixed length, immutable, and ordered container. Elements of tuple are separated by commas and written between parentheses. Examples of tuples:

[43]:
```
(3,)              # a singleton
(1,3)             # a pair
(1, "hello", 1.0); # a triple
```

Note the difference between `(3)` and `(3,)`. Because the parentheses can also be used to group expressions, the first one defines an integer, but the second one defines a tuple with single element.

As we can see, both lists and tuples can contain values of different type.

List, tuples, and strings are called *sequences* in Python, and they have several commonalities:

- their length can be queried with the `len` function
- `min` and `max` function find the minimum and maximum element of a sequence, and `sum` adds all the elements of numbers together
- Sequences can be concatenated with the `+` operator, and repeated with the `*` operator: `"hi"*3=="hihihi"`
- Since sequences are ordered, we can refer to the elements of a sequences by integers using the *indexing* notation: `"abcd"[2] == "c"`
- Note that the indexing begins from 0
- Negative integers start indexing from the end: -1 refers to the last element, -2 refers to the second last, and so on

Above we saw that we can access a single element of a sequence using *indexing*. If we want a subsequence of a sequence, we can use the *slicing* syntax. A slice consists of elements of the original sequence, and it is itself a sequence as well. A simple slice is a range of elements:

[44]:
```
s="abcdefg"
s[1:4]
```

[44]:
```
'bcd'
```

Note that Python ranges exclude the last index. The generic form of a slice is `sequence[first:last:step]`. If any of the three parameters are left out, they are set to default values as follows: first=0, last=len(L), step=1. So, for instance “abcde”[1:]==”bcde”. The step parameter selects elements that are step distance apart from each other. For example:

[45]:
```
print([0,1,2,3,4,5,6,7,8,9][::3])
```

```
[0, 3, 6, 9]
```

Exercise 8 (solve quadratic)

In mathematics, the quadratic equation $ax^2 + bx + c = 0$ can be solved with the formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Write a function `solve_quadratic`, that returns both solutions of a generic quadratic as a pair (2-tuple) when the coefficients are given as parameters. It should work like this:

```
print(solve_quadratic(1,-3,2))
(2.0,1.0)
print(solve_quadratic(1,2,1))
(-1.0,-1.0)
```

You may want to use the `math.sqrt` function from the `math` module in your solution. Test that your function works in the main function!

Modifying lists

We can assign values to elements of a list by indexing or by slicing. An example:

[46]:
```
L=[11,13,22,32]
L[2]=10          # Changes the third element
print(L)
```

```
[11, 13, 10, 32]
```

Or we can assign a list to a slice:

[47]:
```
L[1:3]=[4]
print(L)
```

```
[11, 4, 32]
```

We can also modify a list by using *mutating methods* of the `list` class, namely the methods `append`, `extend`, `insert`, `remove`, `pop`, `reverse`, and `sort`. Try Python's help functionality to find more about these methods: e.g. `help(list.extend)` or `help(list)`.

> Note that we cannot perform these modifications on tuples or strings since they are *immutable*

Generating numerical sequences

Trivial lists can be tedious to write: `[0,1,2,3,4,5,6]`. The function `range` creates numeric ranges automatically. The above sequence can be generated with the function call `range(7)`. Note again that then end value is not included in the sequence. An example of using the `range` function:

[48]:
```
L=range(3)
for i in L:
    print(i)
# Note that L is not a list!
print(L)
```

```
0
1
2
range(0, 3)
```

So `L` is not a list, but it is a sequence. We can for instace access its last element with `L[-1]`. If really needed, then it can be converted to a list with the `list` constructor:

[49]:
```
L=range(10)
print(list(L))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that using a range consumes less memory than the corresponding list. This is because in a list all the elements are stored in the memory, whereas the range generates the requested elements only when needed. For example, when the for loop asks for the next element from the range at each iteration, only a single element from the range exists in memory at the same time. This makes a big difference when using large ranges, like range(1000000).

The `range` function works in similar fashion as slices. So, for instance the step of the sequence can be given:

[50]:
```
print(list(range(0, 7, 2)))
```

```
[0, 2, 4, 6]
```

Sorting sequences

In Python there are two ways to sort sequences. The `sort` *method* modifies the original list, whereas the `sorted` *function* returns a new sorted list and leaves the original intact. A couple of examples will demonstrate this:

[51]:
```
L=[5,3,7,1]
L.sort()      # here we call the sort method of the object L
print(L)
L2=[6,1,7,3,6]
print(sorted(L2))
print(L2)
```

```
[1, 3, 5, 7]
[1, 3, 6, 6, 7]
[6, 1, 7, 3, 6]
```

The parameter `reverse=True` can be given (both to `sort` and `sorted`) to get descending order of elements:

[52]:
```
L=[5,3,7,1]
print(sorted(L, reverse=True))
```

```
[7, 5, 3, 1]
```

Exercise 9 (merge)

Suppose we have two lists `L1` and `L2` that contain integers which are sorted in ascending order. Create a function `merge` that gets these lists as parameters and returns a new sorted list `L` that has all the elements of `L1` and `L2`. So, `len(L)` should equal to `len(L1)+len(L2)`. Do this using the fact that both lists are already sorted. You can't use the `sorted` function or the `sort` method in implementing the `merge` method. You can however use these `sorted` in the main function for creating inputs to the `merge` function. Test with a couple of examples in the `main` function that your solution works correctly.

Note: In Python argument lists are passed by reference to the function, they are not copied! Make sure you don't modify the original lists of the caller.

Exercise 10 (detect ranges)

Create a function named `detect_ranges` that gets a list of integers as a parameter. The function should then sort this list, and transform the list into another list where pairs are used for all the detected intervals. So `3,4,5,6` is replaced by the pair `(3,7)`. Numbers that are not part of any interval result just single numbers. The resulting list consists of these numbers and pairs, separated by commas. An example of how this function works:

```
print(detect_ranges([2,5,4,8,12,6,7,10,13]))
[2,(4,9),10,(12,14)]
```

Note that the second element of the pair does not belong to the range. This is consistent with the way Python's `range` function works. You may assume that no element in the input list appears multiple times.

Zipping sequences

The `zip` function combines two (or more) sequences into one sequence. If, for example, two sequences are zipped together, the resulting sequence contains pairs. In general, if `n` sequences are zipped together, the elements of the resulting sequence contains `n`-tuples. An example of this:

[53]:
```
L1=[1,2,3]
L2=["first", "second", "third"]
print(zip(L1, L2))              # Note that zip does not return a list, like range
print(list(zip(L1, L2)))        # Convert to a list
```

```
<zip object at 0x7fb8141907c8>
[(1, 'first'), (2, 'second'), (3, 'third')]
```

Here's another example of using the `zip` function.

[7]:
```
days="Monday Tuesday Wednesday Thursday Friday Saturday Sunday".split()
weathers="rainy rainy sunny cloudy rainy sunny sunny".split()
temperatures=[10,12,12,9,9,11,11]
for day, weather, temperature in zip(days,weathers,temperatures):
    print(f"On {day} it was {weather} and the temperature was {temperature} degrees celsius.")

# Or equivalently:
#for t in zip(days,weathers,temperatures):
#    print("On {} it was {} and the temperature was {} degrees celsius.".format(*t))
```

```
On Monday it was rainy and the temperature was 10 degrees celsius.
On Tuesday it was rainy and the temperature was 12 degrees celsius.
On Wednesday it was sunny and the temperature was 12 degrees celsius.
On Thursday it was cloudy and the temperature was 9 degrees celsius.
On Friday it was rainy and the temperature was 9 degrees celsius.
On Saturday it was sunny and the temperature was 11 degrees celsius.
On Sunday it was sunny and the temperature was 11 degrees celsius.
```

If the sequences are not of equal length, then the resulting sequence will be as long as the shortest input sequence is.

Exercise 11 (interleave)

Write function `interleave` that gets arbitrary number of lists as parameters. You may assume that all the lists have equal length. The function should return one list containing all the elements from the input lists interleaved. Test your function from the `main` function of the program.

Example: `interleave([1,2,3], [20,30,40], ['a', 'b', 'c'])` should return `[1, 20, 'a', 2, 30, 'b', 3, 40, 'c']`. Use the `zip` function to implement `interleave`. Remember the `extend` method of list objects.

Enumerating sequences

In some other programming languages one iterates through the elements using their indices (0,1, ...) in the sequence. In Python we normally don't need to think about indices when iterating, because the `for` loop allows simpler iteration through the elements. But sometimes you really need to know the index of the current element in the sequence. In this case one uses Python's `enumerate` function. In the next example we would like find the second occurrence of integer 5 in a list.

[55]:
```
L=[1,2,98,5,-1,2,0,5,10]
counter = 0
for i, x in enumerate(L):
    if x == 5:
        counter += 1
        if counter == 2:
            break
print(i)
```

```
7
```

The `enumerate(L)` function call can be thought to be equivalent to `zip(range(len(L)), L)`.

Dictionaries

A *dictionary* is a dynamic, unordered container. Instead of using integers to access the elements of the container, the dictionary uses *keys* to access the stored *values*. The dictionary can be created by listing the comma separated key-value pairs in braces. Keys and values are separated by a colon. A tuple (key,value) is called an *item* of the dictionary.

Let's demonstrate the dictionary creation and usage:

[56]:
```
d={"key1":"value1", "key2":"value2"}
print(d["key1"])
print(d["key2"])
```

```
value1
value2
```

Keys can have different types even in the same container. So the following code is legal: `d={1:"a", "z":1}`. The only restriction is that the keys must be *hashable*. That is, there has to be a mapping from keys to integers. Lists are *not* hashable, but tuples are!

There are alternative syntaxes for dictionary creation:

[57]:
```
dict([("key1", "value1"), ("key2", "value2"), ("key3", "value3")]) # list of items
dict(key1="value1", key2="value2", key3="value3");
```

If a key is not found in a dictionary, the indexing `d[key]` results in an error (*exception* `KeyError`). But an assignment with a non-existing key causes the key to be added in the dictionary associated with the corresponding value:

[58]:
```
d={}
d[2]="value"
print(d)
```

```
{2: 'value'}
```

[59]:
```
# d[1]   # This would cause an error
```

Dictionary object contains several non-mutating methods:

```
d.copy()
d.items()
d.keys()
d.values()
d.get(k[,x])
```

Some methods mutate the dictionary:

```
d.clear()
d.update(d1)
d.setdefault(k[,x])
d.pop(k[,x])
d.popitem()
```

Try out some of these in the below cell. You can find more info with `help(dict)` or `help(dict.keys)`.

[60]:
```
d=dict(a=1, b=2, c=3, d=4, e=5)
d.values()
```

[60]:
```
dict_values([1, 2, 3, 4, 5])
```

Sets

Set is a dynamic, unordered container. It works a bit like dictionary, but only the keys are stored. And each key can be stored only once. The set requires that the keys to be stored are hashable. Below are a few ways of creating a set:

[61]:
```
s={1,1,1}
print(s)
s=set([1,2,2,'a'])
print(s)
s=set()  # empty set
print(s)
s.add(7) # add one element
print(s)
```

```
{1}
{1, 2, 'a'}
set()
{7}
```

A more useful example:

[8]:
```
s="mississippi"
print(f"There are {len(set(s))} distinct characters in {s}")
```

```
There are 4 distinct characters in mississippi
```

The `set` provides the following non-mutating methods:

[63]:
```
s=set()
s1=set()
s.copy()
s.issubset(s1)
s.issuperset(s1)
s.union(s1)
s.intersection(s1)
s.difference(s1)
s.symmetric_difference(s1);
```

The last four operation can be tedious to write to create a more complicated expression. The alternative is to use the corresponding operator forms: `|`, `&`, `-`, and `^`. An example of these:

[64]:
```
s=set([1,2,7])
t=set([2,8,9])
print("Union:", s|t)
print("Intersection:", s&t)
print("Difference:", s-t)
print("Symmetric difference", s^t)
```

```
Union: {1, 2, 7, 8, 9}
Intersection: {2}
Difference: {1, 7}
Symmetric difference {1, 7, 8, 9}
```

There are also the following mutating methods:

```
s.add(x)
s.clear()
s.discard()
s.pop()
s.remove(x)
```

And the set operators `|`, `&`, `-`, and `^` have the corresponding mutating, augmented assignment forms: `|=`, `&=`, `-=`, and `^=`.

Exercise 12 (distinct characters)

Write function `distinct_characters` that gets a list of strings as a parameter. It should return a dictionary whose keys are the strings of the input list and the corresponding values are the numbers of distinct characters in the key.

Use the `set` container to temporarily store the distinct characters in a string. Example of usage: `distinct_characters(["check", "look", "try", "pop"])` should return `{ "check" : 4, "look" : 3, "try" : 3, "pop" : 2}`.

Miscellaneous stuff

To find out whether a container includes an element, the `in` operator can be used. The operator returns a truth value. Some examples of the usage:

[65]:
```
print(1 in [1,2])
d=dict(a=1, b=3)
print("b" in d)
s=set()
print(1 in s)
print("x" in "text")
```

```
True
True
False
True
```

As a special case, for strings the `in` operator can be used to check whether a string is part of another string:

[66]:
```
print("issi" in "mississippi")
print("issp" in "mississippi")
```

```
True
False
```

Elements of a container can be unpacked into variables:

[10]:
```
first, second = [4,5]
a,b,c = "bye"
print(c)
d=dict(a=1, b=3)
key1, key2 = d
print(key1, key2)
```

```
e
a b
```

In membership testing and unpacking only the keys of a dictionary are used, unless either values or items (like below) are explicitly asked.

[11]:
```
for key, value in d.items():
    print(f"For key '{key}' value {value} was stored")
```

```
For key 'a' value 1 was stored
For key 'b' value 3 was stored
```

To remove the binding of a variable, use the `del` statement. For example:

[69]:
```
s="hello"
del s
# print(s)    # This would cause an error
```

To delete an item from a container, the `del` statement can again be applied:

[70]:
```
L=[13,23,40,100]
del L[1]
print(L)
```

```
[13, 40, 100]
```

In similar fashion `del` can be used to delete a slice. Later we will see that `del` can delete attributes from an object.

Exercise 13 (reverse dictionary)

Let `d` be a dictionary that has English words as keys and a list of Finnish words as values. So, the dictionary can be used to find out the Finnish equivalents of an English word in the following way:

```
d["move"]
["liikuttaa"]
d["hide"]
["piilottaa", "salata"]
```

Make a function `reverse_dictionary` that creates a Finnish to English dictionary based on a English to Finnish dictionary given as a parameter. The values of the created dictionary should be lists of words. It should work like this:

```
d={'move': ['liikuttaa'], 'hide': ['piilottaa', 'salata'], 'six': ['kuusi'], 'fir': ['kuusi']}
reverse_dictionary(d)
{'liikuttaa': ['move'], 'piilottaa': ['hide'], 'salata': ['hide'], 'kuusi': ['six', 'fir']}
```

Be careful with synonyms and homonyms!

Exercise 14 (find matching)

Write function `find_matching` that gets a list of strings and a search string as parameters. The function should return the indices to those elements in the input list that contain the search string. Use the function `enumerate`.

An example: `find_matching(["sensitive", "engine", "rubbish", "comment"], "en")` should return the list `[0, 1, 3]`.

Compact way of creating data structures

We can now easily create complicated data structures using `for` loops:

[71]:
```
L=[]
for i in range(10):
    L.append(i**2)
print(L)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Because this kind of pattern is often used, Python offers a short-hand for this. A *list comprehension* is an expression that allows creating complicated lists on one line. The notation is familiar from mathematics:

$\{a^3 : a \in \{1, 2, \ldots, 10\}\}$

The same written in Python as a list comprehension:

[72]:
```
L=[ a**3 for a in range(1,11)]
print(L)
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

The generic form of a list comprehension is: `[ expression for element in iterable lc-clauses ]`. Let's break this syntax into pieces. The iterable can be any sequence (or something more general). The lc-clauses consists of zero or more of the following clauses:

- for elem in iterable
- if expression

A more complicated example. How would you describe these numbers?

[73]:
```
L=[ 100*a + 10*b +c for a in range(0,10)
                    for b in range(0,10)
                    for c in range(0,10)
                    if a <= b <= c]
print(L)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 22, 23, 24, 25, 26, 27, 28,
29, 33, 34, 35, 36, 37, 38, 39, 44, 45, 46, 47, 48, 49, 55, 56, 57, 58, 59, 66, 67, 68, 69, 77,
78, 79, 88, 89, 99, 111, 112, 113, 114, 115, 116, 117, 118, 119, 122, 123, 124, 125, 126, 127,
128, 129, 133, 134, 135, 136, 137, 138, 139, 144, 145, 146, 147, 148, 149, 155, 156, 157, 158,
159, 166, 167, 168, 169, 177, 178, 179, 188, 189, 199, 222, 223, 224, 225, 226, 227, 228, 229,
233, 234, 235, 236, 237, 238, 239, 244, 245, 246, 247, 248, 249, 255, 256, 257, 258, 259, 266,
267, 268, 269, 277, 278, 279, 288, 289, 299, 333, 334, 335, 336, 337, 338, 339, 344, 345, 346,
347, 348, 349, 355, 356, 357, 358, 359, 366, 367, 368, 369, 377, 378, 379, 388, 389, 399, 444,
445, 446, 447, 448, 449, 455, 456, 457, 458, 459, 466, 467, 468, 469, 477, 478, 479, 488, 489,
499, 555, 556, 557, 558, 559, 566, 567, 568, 569, 577, 578, 579, 588, 589, 599, 666, 667, 668,
669, 677, 678, 679, 688, 689, 699, 777, 778, 779, 788, 789, 799, 888, 889, 899, 999]
```

If one needs only to iterate through the list once, it is more memory efficient to use a *generator expression* instead. The only thing that changes syntactically is that the surrounding brackets are replaced by parentheses:

[74]:
```
G = ( 100*a + 10*b + c for a in range(0,10)
                       for b in range(0,10)
                       for c in range(0,10)
                       if a <= b <= c )
print(sum(G))   # This iterates through all the elements from the generator
print(sum(G))   # It doesn't restart from the beginning, so all elements are already consumed
```

```
60885
0
```

Note above that one can only iterate through the generator once.

Similary a *dictionary comprehension* creates a dictionary:

[75]:
```
d={ k : k**2 for k in range(10)}
print(d)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

And a *set comprehension* creates a set:

[76]:
```
s={ i*j for i in range(10) for j in range(10)}
print(s)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 35, 36,
40, 42, 45, 48, 49, 54, 56, 63, 64, 72, 81}
```

Exercise 15 (two dice comprehension)

Redo the earlier exercise which printed all the pairs of two dice results that sum to 5. But this time use a list comprehension. Print one pair per line.

Processing sequences

In this section we will go through some useful tools, that are maybe familiar to you from some functional programming language like *lisp* or *haskell*. These functions rely on functions being first-class objects in Python, that is, you can

- pass a function as a parameter to another function
- return a function as a return value from some function
- store a function in a data structure or a variable

We will talk about `map`, `filter`, and `reduce` functions. We will also cover how to create functions with no name using the *lambda* expressions.

Map and lambda functions

The `map` function gets a list and a function as parameters, and it returns a new list whose elements are elements of the original list transformed by the parameter function. For this to work the parameter function must take exactly one value in and return a value out. An example will clarify this concept:

[77]:
```
def double(x):
    return 2*x
L=[12,4,-1]
print(map(double, L))
```

```
<map object at 0x7fb81413ef60>
```

The map function returns a map object for efficiency reasons. However, since we only want print the contents, we first convert it to a list and then print it:

[78]:
```
print(list(map(double,L)))
```

```
[24, 8, -2]
```

When one reads numeric data from a file or from the internet, the numbers are usually in string form. Before they can be used in computations, they must first be converted to ints or floats. A simple example will showcase this.

[79]:
```
s="12 43 64 6"
L=s.split()        # The split method of the string class, breaks the string at whitespaces
                   # to a list of strings.
print(L)
print(sum(map(int, L)))  # The int function converts a string to an integer
```

```
['12', '43', '64', '6']
125
```

Sometimes it feels unnecessary to write a function is you are only going to use it in one `map` function call. For example the function

[80]:
```
def add_double_and_square(x):
    return 2*x+x**2
```

It is not likely that you will need it elsewhere in your program. The solution is to use an *expression* called *lambda* to define a function with no name. Because it is an expression, we can put it, for instance, in an argument list of a function call. The lambda expression has the form `lambda param1,param2, ... : expression`, where after the lambda keyword you list the parameters of the function, and after the colon is the expression that uses the parameters to compute the return value of the function. Let's replace the above `add_double_and_square` function with a lambda function and apply it to a list using the `map` function.

[81]:
```
L=[2,3,5]
print(list(map(lambda x : 2*x+x**2, L)))
```

```
[8, 15, 35]
```

Exercise 16 (transform)

Write a function `transform` that gets two strings as parameters and returns a list of integers. The function should split the strings into words, and convert these words to integers. This should give two lists of integers. Then the function should return a list whose elements are multiplication of two integers in the respective positions in the lists. For example `transform("1 5 3", "2 6 -1")` should return the list of integers `[2, 30, -3]`.

You **have** to use `split`, `map`, and `zip` functions/methods. You may assume that the two input strings are in correct format.

Filter function

The `filter` function takes a function and a list as parameters. But unlike with the map construct, now the parameter function must take exactly one parameter and return a truth value (True or False). The `filter` function then creates a new list with only those elements from the original list for which the parameter function returns True. The elements for which the parameter function returns False are filtered out. An example will demonstrate the `filter` function:

[1]:
```
def is_odd(x):
    """Returns True if x is odd and False if x is even"""
    return x % 2 == 1         # The % operator returns the remainder of integer division
L=[1, 4, 5, 9, 10]
print(list(filter(is_odd, L)))
```

```
[1, 5, 9]
```

The even elements of the list were filtered out.

Note that the `filter` function is rarely used in modern python since list comprehensions can do the same thing while also doing whatever we want to do with the filtered values.

[3]:
```
[l**2 for l in L if is_odd(l)] # squares of odd values
```

[3]:
```
[1, 25, 81]
```

That said, `filter` is a useful function to know.

Exercise 17 (positive list)

Write a function `positive_list` that gets a list of numbers as a parameter, and returns a list with the negative numbers and zero filtered out using the `filter` function.

The function call `positive_list([2,-2,0,1,-7])` should return the list `[2,1]`. Test your function in the `main` function.

The reduce function

The `sum` function that returns the sum of a numeric list, can be though to reduce a list to a single element. It does this reduction by repeatedly applying the `+` operator until all the list elements are consumed. For instance, the list `[1,2,3,4]` is reduced by the expression `(((0+1)+2)+3)+4` of repeated applications of the `+` operator. We could implement this with the following function:

[83]:
```
def sumreduce(L):
    s=0
    for x in L:
        s = s+x
    return s
```

Because this is a common pattern, the `reduce` function is a common inclusion in functional programming languages. In Python `reduce` is included in the `functools` module. You give the operator you want to use as a parameter to reduce (addition in the above example). You may also give a starting value of the computation (starting value 0 was used above).

If no starting value is used, the first element of the iterable is used as the starting value.

We can now get rid of the separate function `sumreduce` by using the reduce function:

[2]:
```
L=[1,2,3,4]
from functools import reduce   # import the reduce function from the functools module
reduce(lambda x,y:x+y, L, 0)
```

[2]:
```
10
```

If we wanted to get a product of all numbers in a sequence, we would use

[3]:
```
reduce(lambda x,y:x*y, L, 1)
```

[3]:
```
24
```

This corresponds to the sequence `(((1*1)*2)*3)*4` of application of operator `*`.

Note that use of the starting value is necessary, because we want to be able to reduce lists of length 0 as well. If no starting value is specified when run on an empty list, reduce will raise an exception.

String handling

We have already seen how to index, slice, concatenate, and repeat strings. Let's now look into what methods the `str` class offers. In Python strings are immutable. This means that for instance the following assignment is not legal:

[86]:
```
s="text"
# s[0] = "a"    # This is not legal in Python
```

Because of the immutability of the strings, the string methods work by returning a value; they don't have any side-effects. In the rest of this section we briefly describe several of these methods. The methods are here divided into five groups.

Classification of strings

All the following methods will take no parameters and return a truth value. An empty string will always result in `False`.

- `s.isalnum()` True if all characters are letters or digits
- `s.isalpha()` True if all characters are letters
- `s.isdigit()` True if all characters are digits
- `s.islower()` True if contains letters, and all are lowercase
- `s.isupper()` True if contains letters, and all are uppercase
- `s.isspace()` True if all characters are whitespace
- `s.istitle()` True if uppercase in the beginning of word, elsewhere lowercase

String transformations

The following methods do conversions between lower and uppercase characters in the string. All these methods return a new string.

- `s.lower()` Change all letters to lowercase
- `s.upper()` Change all letters to uppercase
- `s.capitalize()` Change all letters to capitalcase
- `s.title()` Change to titlecase
- `s.swapcase()` Change all uppercase letters to lowercase, and vice versa

Searching for substrings

All the following methods get the wanted substring as the parameter, except the replace method, which also gets the replacing string as a parameter

- `s.count(substr)` Counts the number of occurences of a substring
- `s.find(substr)` Finds index of the first occurence of a substring, or -1
- `s.rfind(substr)` Finds index of the last occurence of a substring, or -1
- `s.index(substr)` Like find, except ValueError is raised if not found
- `s.rindex(substr)` Like rfind, except ValueError is raised if not found

- `s.startswith(substr)` Returns True if string starts with a given substring
- `s.endswith(substr)` Returns True if string ends with a given substring
- `s.replace(substr, replacement)` Returns a string where occurences of one string are replaced by another

Keep also in mind that the expression `"issi" in "mississippi"` returns a truth value of whether the first string occurs in the second string.

Trimming and adjusting

- `s.strip(x)` Removes leading and trailing whitespace by default, or characters found in string x
- `s.lstrip(x)` Same as strip but only leading characters are removed
- `s.rstrip(x)` Same as strip but only trailing characters are removed
- `s.ljust(n)` Left justifies string inside a field of length n
- `s.rjust(n)` Right justifies string inside a field of length n
- `s.center(n)` Centers string inside a field of length n

An example of using the `center` method and string repetition:

[12]:
```
L=[1,3,5,7,9,1,1]
print("-"*11)
for i in L:
    s="*"*i
    print(f"|{s.center(9)}|")
print("-"*11)
```

```
-----------
|    *    |
|   ***   |
|  *****  |
| ******* |
|*********|
|    *    |
|    *    |
-----------
```

Joining and splitting

The `join(seq)` method joins the strings of the sequence `seq`. The string itself is used as a delimitter. An example:

[88]:
```
"--".join(["abc", "def", "ghi"])
```

[88]:
```
'abc--def--ghi'
```

[5]:
```
L=[str(x) for x in range(100)]
s=""
for x in L:
    s += " " + x    # Avoid doing this, it creates a new string at every iteration
print(s)            # Note the redundant initial space
print(" ".join(L))  # This is the correct way of building a string out of smaller strings
```

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
99
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

If you want to build a string out of smaller strings, then first put the small strings into a list, and then use the `join` method to catenate the pieces together. It is much more efficient this way. Use the + catenation operator only if you have very few short strings that you want to catenate.

Below we can see that for our small (100 element) list, execution is an order of magnitude faster using the `join` method.

[6]:
```
%%timeit
s=""
for x in L:
    s += " " + x
```

```
11.8 µs ± 114 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[8]:
```
%%timeit
s = " ".join(L)
```

```
1.25 µs ± 6.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

`%%timeit` is an IPython cell magic command, that is useful for timing execution in notebooks.

The method `split(sep=None)` divides a string into pieces that are separated by the string `sep`. The pieces are returned in a list. For instance, the call `'abc--def--ghi'.split("--")` will result in

[90]:
```
'abc--def--ghi'.split("--")
```

[90]:
```
['abc', 'def', 'ghi']
```

If no parameters are given to the `split` method, then it splits at any sequence of white space.

Exercise 18 (acronyms)

Write function `acronyms` which takes a string as a parameter and returns a list of acronyms. A word is an acronym if it has length at least two, and all its characters are in uppercase. Before acronym detection, delete punctuation with the `strip` method.

Test this function in the `main` function with the following call:

```
print(acronyms("""For the purposes of the EU General Data Protection Regulation (GDPR), the
controller of your personal information is International Business Machines Corporation (IBM Corp.), 1
New Orchard Road, Armonk, New York, United States, unless indicated otherwise. Where IBM Corp. or a
subsidiary it controls (not established in the European Economic Area (EEA)) is required to appoint a
legal representative in the EEA, the representative for all such cases is IBM United Kingdom Limited,
PO Box 41, North Harbour, Portsmouth, Hampshire, United Kingdom PO6 3AU."""))
```

This should return `['EU', 'GDPR', 'IBM', 'IBM', 'EEA', 'EEA', 'IBM', 'PO', 'PO6', '3AU']`

Exercise 19 (sum equation)

Write a function `sum_equation` which takes a list of positive integers as parameters and returns a string with an equation of the sum of the elements.

Example: `sum_equation([1,5,7])` returns `"1 + 5 + 7 = 13"` Observe, the spaces should be exactly as shown above. For an empty list the function should return the string “0 = 0”.

Modules

To ease management of large programs, software is divided into smaller pieces. In Python these pieces are called *modules*. A module should be a unit that is as independent from other modules as possible. Each file in Python corresponds to a module. Modules can contain classes, objects, functions, ... For example, functions to handle regular expressions are in module `re`

The standard library of Python consists of hundreds of modules. Some of the most common standard modules include

- `re`
- `math`
- `random`
- `os`
- `sys`

Any file with extension `.py` that contains Python source code is a module. So, no special notation is needed to create a module.

Using modules

Let's say that we need to use the cosine function. This function, and many other mathematical functions are located in the `math` module. To tell Python that we want to access the features offered by this module, we can give the statement `import math`. Now the module is loaded into memory. We can now call the function like this:

```
math.cos(0)
1.0
```

Note that we need to include the module name where the `cos` function is found. This is because other modules may have a function (or other attribute of a module) with the same name. This usage of different namespace for each module prevents name clashes. For example, functions `gzip.open`, `os.open` are not to be confused with the builtin `open` function.

Breaking the namespace

If the cosine is needed a lot, then it might be tedious to always specify the namespace, especially if the name of the namespace/module is long. For these cases there is another way of importing modules. Bring a name to the current scope with `from math import cos` statement. Now we can use it without the namespace specifier: `cos(1)`.

Several names can be imported to the current scope with `from math import name1, name2, ...` Or even all names of the module with `from math import *` The last form is sensible only in few cases, normally it just confuses things since the user may have no idea what names will be imported.

Module lookup

When we try to import a module `mod` with the import statement, the lookup proceeds in the following order:

- Check if it is a builtin module
- Check if the file `mod.py` is found in any of the folders in the list `sys.path`. The first item in this list is the current folder

When Python is started, the `sys.path` list is initialised with the contents of the `PYTHONPATH` environment variable

Module hierarchy

The standard library contains hundreds of modules. Hence, it is hard to comprehend what the library includes. The modules therefore need to be organised somehow. In Python the modules can be organised into hierarchies using *packages*. A package is a module that can contain other packages and modules. For example, the `numpy` package contains subpackages `core`, `distutils`, `f2py`, `fft`, `lib`, `linalg`, `ma`, `numarray`, `oldnumeric`, `random`, and `testing`. And package `numpy.linalg` in turn contains modules `linalg`, `lapack_lite` and `info`.

Importing from packages

The statement `import numpy` imports the top-level package `numpy` and its subpackages.

- `import numpy.linalg` imports the subpackage only, and
- `import numpy.linalg.linalg` imports the module only

If we want to skip the long namespace specification, we can use the form

```
from numpy.linalg import linalg
```

or

```
from numpy.linalg import linalg as lin
```

if we want to use a different name for the module. The following command imports the function `det` (computes the determinant of a matrix) from the module linalg, which is contained in a subpackage linalg, which belongs to package numpy:

```
from numpy.linalg.linalg import det
```

Had we only imported the top-level package `numpy` we would have to refer to the `det` function with the full name `numpy.linalg.linalg.det`.

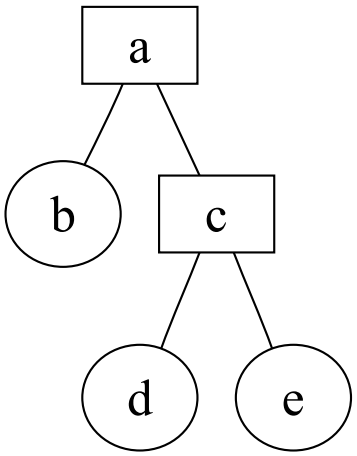
Here's a recap of the module hierarchy:

```
numpy    package
  .
linalg   subpackage
  .
linalg   module
  .
 det     function
```

Correspondence between folder and module hierarchies

The packages are represented by folders in the filesystem. The folder should contain a file named `__init__.py` that makes up the package body. This handles the initialisation of the package. The folder may contain also further folders (subpackages) or Python files (normal modules).

```
a/
    __init__.py
    b.py
    c/
        __init__.py
        d.py
        e.py
```



Contents of a module

Suppose we have a module named `mod.py`. All the assignments, class definitions with the `class` statement, and function definitions with `def` statement will create new attributes to this module. Let's import this module from another Python file using the `import mod` statement. After the import we can access the attributes of the module object using the normal dot notation: `mod.f()`, `mod.myclass()`, `mod.a`, etc. Note that Python doesn't really have global variables that are visible to all modules. All variables belong to some module namespace.

One can query the attributes of an object using the `dir` function. With no parameters, it shows the attributes of the current module. Try executing `dir()` in an IPython shell or in a Jupyter notebook! After that, define the following attributes, and try running `dir()` again:

```
a=5
def f(i):
    return i + 1
```

The above definitions created a *data attribute* called `a` and a *function attribute* called `f`. We will talk more about attributes next week when we will talk about objects.

Just like other objects, the module object contains its attributes in the dictionary `modulename.__dict__` Usually a module contains at least the attributes `__name__` and `__file__`. Other common attributes are `__version__`, `__author__` and `__doc__`, which contains the docstring of the module. If the first statement of a file is a string, this is taken as the docstring for that module. Note that the docstring of the module really must be the first non-empty non-comment line. The attribute `__file__` is always the filename of the module.

The module attribute `__name__` has value `"__main__"` if we in are the main program, otherwise some other module has imported us and name equals `__file__`.

In Python it is possible to put statements on the top-level of our module `mod` so that they don't belong to any function. For instance like this:

```
for _ in range(3):
    print("Hello")
```

But if somebody imports our module with `import mod`, then all the statements at the top-level will be executed. This may be surprising to the user who imported the module. The user will usually say, explicitly when he/she wants to execute some code from the imported module.

It is better style to put these statements inside some function. If they don't fit in any other function, then you can use, for example, the function named `main`, like this:

```
def main():
    for _ in range(3):
        print("Hello")

if __name__ == "__main__":    # We call main only when this module is not being imported, but
directly executed
    main()                    # for example with 'python3 mod.py'
```

You probably have seen this mechanism used in the exercise stubs. Note that in Python the `main` has no special meaning, it is just our convention to use it here. Now if somebody imports `mod`, the `for` loop won't be automatically executed. If we want, we can call it explicitly with `mod.main()`.

```
for _ in range(3):
    print("Hello")
```

Exercise 20 (usemodule)

Create your own module as file `triangle.py` in the `src` folder. The module should contain two functions:

- `hypothenuse` which returns the length of the hypothenuse when given the lengths of two other sides of a right-angled triangle
- `area` which returns the area of the right-angled triangle, when two sides, perpendicular to each other, are given as parameters.

Make sure both the functions and the module have descriptive docstrings. Add also the `__version__` and `__author__` attributes to the module. Call both your functions from the main function (which is in file `usemodule.py`).

Summary

- We have learned that Python's code blocks are denoted by consistent indenting, with spaces or tabs, unlike in many other languages
- Python's `for` loops goes through all the elements of a container without the need of worrying about the positions (indices) of the elements in the container
- More generally, an iterable is an object whose elements can be gone through one by one using a `for` loop. Such as `range(1,7)`
- Python has dynamic typing: the type of a name is known only when we run the program. The type might not be fixed, that is, if a name is created, for example, in a loop, then its type might change at each iteration.
- Visibility of a name: a name that refers to a variable can disappear in the middle of a code block, if a `del` statement is issued!
- Python is good at string handling, but remember that if you want to concatenate large number of strings, use the `join` method. Concatenating by the `+` operator multiple times is very inefficient
- Several useful tools exist to process sequences: `map`, `reduce`, `filter`, `zip`, `enumerate`, and `range`. The unnamed lambda function can be helpful with these tools. Note that these tools (except the `reduce`) don't return lists, but iterables, for efficiency reasons: Most often we don't want to store the result from these tools to a container (such as a list), we may only want to iterate through the result!