Introduction

September 30, 2022

A Jupyter notebook consists of two different types of cells: - *Code cells* contain Python code, while - *Markdown cells* are used to write comments.

The current cell is a *markdown cell*, the next cell is a *code cell*. This code cell is only needed to format this notebook.

[1]:
```
from IPython.core.display import HTML
with open('../style.css', 'r') as file:
    css = file.read()
HTML(css)
```

[1]:
```
<IPython.core.display.HTML object>
```

1 An Introduction to *Python*

Let us begin by checking the version of our Python interpreter. The package `sys` contains the variable `version` that stores this information.

[2]:
```
import sys
sys.version
```

[2]:
```
'3.10.4 (main, Mar 31 2022, 03:38:35) [Clang 12.0.0 ]'
```

This notebook gives a short introduction to *Python*. We will just present the basics. More interesting features will be presented later.

1.1 Evaluating expressions

As Python is an interactive language, expressions can be evaluated directly. In a *Jupyter* notebook we just have to type Ctrl-Enter in the cell containing the expression. Instead of Ctrl-Enter we can also use Shift-Enter.

[3]:
```
1 + 2
```

[3]:
```
3
```

In *Python*, the precision of integers is not bounded. Hence, the following expression does **not** cause an integer overflow.

[4]:
```
1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11 * 12 * 13 * 14 * 15 * 16 * 17 * 18 *␣
↪19 * 20 * 21 * 22
```

[4]:
```
1124000727777607680000
```

Here is another example that show that *Python* is able to compute with huge numbers.
In this example, the operator `**` is used for exponentiation, i.e. the expression

```
7 ** 125
```

is interpreted as 7^{125}.

[5]:
```
7 ** 125
```

[5]:
```
43376549480979932825373547572631882516978329946204051017448930177445694327209941
6808967219221175890932080 7
```

In order to print numbers or strings, we can use the function `print`. This function can print objects of any type. In the following example, this function prints a *string*. In *Python* any character sequence enclosed in single quotes is interpreted as a *string*.

[6]:
```
print('Hello, World!')
```

```
Hello, World!
```

The last expression in every notebook cell is automatically printed. Therefore, printing `'Hello, World!'` can also be done by just executing the following cell.

[7]:
```
'Hello, World!'
```

[7]:
```
'Hello, World!'
```

If we want to supress the last expression from being printed, we can end the expression with a semicolon:

[8]:
```
'Hello, World!';
```

Instead of using single quotes we can also use double quotes as seen in the next example. However, the convention is to use single quotes, unless the string itself contains a single quote.

[9]:
```
"Hello, World!"
```

[9]:
```
'Hello, World!'
```

[10]:
```
"This can't be the case!"
```

[10]:
```
"This can't be the case!"
```

The function print accepts any number of arguments. For example, to print the string "36 * 37 / 2 =" followed by the value of the expression $36 \cdot 37/2$ we can use the following print statement:

[11]:
```
print('36 * 37 / 2 =', 36 * 37 // 2)
```

```
36 * 37 / 2 = 666
```

In the expression "36 * 37 // 2" we have used the operator "//" in order to enforce *integer division*, which is also known as *Euclidean division*. If we had used the operator "/" instead, *Python* would have used *floating point division* and therefore it would have printed the floating point number 666.0 instead of the integer 666.

[12]:
```
print('36 * 37 / 2 =', 36 * 37 / 2)
```

```
36 * 37 / 2 = 666.0
```

The next cell demonstrates Euclidean division.

[13]:
```
7 // 3
```

[13]: 2

The *remainder* that is left in an integer division is computed with the operator `%`.

[14]:
```
7 % 3
```

[14]: 1

The result of a computation can be stored in a *variable*. For example, to store the quotient resulting from dividing 7 by 3 into the variable with name `q` and to store the remainder into a variable with name `r` we can write:

[15]:
```
q = 7 // 3
r = 7 % 3
```

Now `q` and `r` contain the quotient and the remainder of the integer division of `7` by `3`as can be seen below:

[16]:
```
q
```

[16]: 2

[17]:
```
r
```

[17]: 1

1.2 Lists

In order to present more interesting examples we next show how *Python* supports *lists*. A list is created using the square brackets `[` and `]`. For example, the empty list, that is the list containing no elements, is written as `[]`.

[18]:
```
[]
```

[18]:
```
[]
```

The list containing the elements `1`, `2`, and `3` is written by seperating the elements with a `,` as shown below:

[19]:
```
[1, 2, 3]
```

[19]:
```
[1, 2, 3]
```

The spaces that I have used in the previous example are optional but recommended.

There are a number of predefined functions that take lists as arguments. For example, the function `sum` takes a list of numbers and adds these numbers.

[20]:
```
sum([1, 2, 3])
```

[20]:
```
6
```

To create a list containing all numbers starting from a given number a up to but excluding the number b we can use *list comprehension* together with the function `range`. For example, the list of the first 10 positive natural numbers can be written as:

[21]:
```
L = [n for n in range(1, 10+1)]
L
```

[21]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The function call `range(a, b)` takes two integers `a` and `b` and returns the sequence of numbers from `a` upto `b-1`.

The general syntax for a list comprehension expression is

$$[x \quad \texttt{for}\ x\ \texttt{in}\ e]$$

where `for` and `in` are keywords, x is a variable used for iterating while e is an expression that evaluates to a so called *iterable*. For example, a list is an *iterable* because we can *iterate* over its elements.

We can also combine *list selection* with *list comprehension*. This is done by using the keyword `if` followed by an expression that evaluates as either `True` of `False` for all elements that are generated by the comprehension expression. For example, the following expression computes the list of all positive even numbers less or equal than 10.

[22]:
```
E = [n for n in L if n % 2 == 0]
E
```

[22]:
```
[2, 4, 6, 8, 10]
```

Here, the expression `n % 2 == 0` tests whether the remainder that is left when `n` is divided by `2` is equal to `0`. Note that we have to use the operator `==` to compare to numbers for equality, since the string `=` is used as the *assignment operator* in *Python*.

The *index operator* $[\cdot]$ is a postfix operator that is used to return the element that is stored at a given *index* in a list. For example, If `L` is a list, then `L[0]` returns the first element of `L`. (Yes, indexing is zero-based in *Python*, so the first element has index `0`.)

[23]:
```
L
```

[23]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[24]:
```
L[0]
```

[24]:
```
1
```

To change the first element of a list, we can use the *index operator* on the left hand side of an *assignment*:

[25]:
```
L[0] = 7
L
```

[25]:
```
[7, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Lists can be *concatenated* with the operator `+`:

[26]:
```
[1,2,3] + [4,5,6]
```

[26]:
```
[1, 2, 3, 4, 5, 6]
```

The function `len` computes the *length* of a list. The length of a list L is defined as the number of elements of L:

[27]:
```
len([4, 5, 4])
```

[27]:
```
3
```

Lists support the functions max and min. The expression `max`(L) computes the maximum of all the elements of the list (or tuple) L, while `min`(L) computes the smallest element of L. These functions also operate on tuples or sets.

[28]:
```
max([1, 2, 3])
```

[28]:
```
3
```

[29]:
```
min([1, 2, 3])
```

[29]:
```
1
```

1.3 Scripts and Functions

Next, we show how it is possible to combine multiple expressions in a *script* and how to create *user defined functions*.

To begin with we present a script that asks the user to input a natural number n and then proceeds to compute the sum of all natural numbers from 1 upto and including n, that is it computes the sum

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

The script contained in the cell below works as follows: * The function input prompts the user to enter a string. * This string is then converted into an integer using the function int. * Next, the list `L` is created such that

$$\texttt{L} = [1, 2, 3, \cdots, (n-2), (n-1), n].$$

* The `print` statement uses the function `sum` to compute the sum of all the elements of the List `L`. The effect of the last argument `sep=''` is to prevent `print` from separating its arguments by spaces.

[30]:
```
n = input('Type a natural number and press return: ')
n = int(n)
L = [i for i in range(1, n+1)]
print('The sum 1 + 2 + ... + ', n, ' is equal to ', sum(L), '.', sep='')
```

```
Type a natural number and press return: 36
The sum 1 + 2 + ... + 36 is equal to 666.
```

If we input the number 36, which happens to be equal to 6^2, then we discover the following remarkable identity:

$$\sum_{i=1}^{6^2} i = 666.$$

The next example shows how *functions* can be defined in *Python*. The function $\texttt{sum}(n)$ is supposed to compute the sum of all the numbers in the set $\{1, \cdots, n\}$. Therefore, we have

$$\texttt{sum}(n) = \sum_{i=1}^{n} i.$$

The function `sum` is defined *recursively*. The recursive implementation of the function `sum` can best by understood if we observe that it satisfies the following two equations: * $\texttt{sum}(0) = 0$, * $\texttt{sum}(n) = \texttt{sum}(n-1) + n$ provided that $n > 0$.

The keyword `def` is used to signal that we define a function.

[31]:
```
def sum(n):
    if n == 0:
        return 0
    return sum(n-1) + n
```

Lets test this function.

[32]:
```
sum(3)
```

[32]: 6

Let us discuss the implementation of the function `sum` line by line: 1. The keyword `def` starts the definition of the function. It is followed by the name of the function that is defined. The name is followed by the list of the parameters of the function. This list is enclosed in parentheses. If there had been more than one parameter, the parameters would have been separated by commas. Finally, there needs to be a colon at the end of the first line. 2. The body of the function is *indented*. **Contrary** to most other programming languages, Python is space sensitive. The first statement of the body is a *conditional statement*, which starts with the keyword `if`. The keyword is followed by a test. In this case we test whether the variable n is equal to the number 0. Note that this test is followed by a colon `:`. 3. The next line contains a `return` statement. Note that this statement is again indented. All statements indented by the same amount that follow an `if`-statement are considered as the *body* of the `if`-statement. In this case the body contains only a single statement. 4. The last line of the function definition contains the recursive invocation of the function `sum` that computes the value `sum(n)` by reducing it to the value `sum(n-1)`.

Using the function `sum`, we can compute the sum $\sum_{i=1}^{n} i$ as follows:

[33]:
```
n     = int(input("Enter a natural number: "))
total = sum(n)
if n > 2:
    print("0 + 1 + 2 + ... + ", n, " = ", total, sep='')
else:
    print(total)
```

```
Enter a natural number: 10
0 + 1 + 2 + ... + 10 = 55
```

1.4 Boolean Values and Boolean Operators

In *Python*, the *truth values*, also known as *Boolean values*, are written as True and False.

[34]:
```
True
```

[34]:
```
True
```

[35]:
```
False
```

[35]:
```
False
```

The following function is needed for pretty-printing. It assumes that the argument val is a truth value. This truth value is then turned into a string that has a size of exactly 5 characters.

[36]:
```
def toStr(val):
    if val:
        return 'True '
    return 'False'
```

These values can be combined using the *Boolean operators* $\wedge$, $\vee$, and $\neg$. In *Python*, these operators are denoted as `and`, `or`, and `not`. The following table shows how the operator `and` is defined:

[37]:
```
B = [True, False]
for x in B:
    for y in B:
        print(toStr(x), 'and', toStr(y), '=', x and y)
```

```
True  and True  = True
True  and False = False
False and True  = False
False and False = False
```

The *disjunction* of two Boolean values is only *False* if both values are *False*. In *Python*, the disjunction operator is written as `or`:

[38]:
```
for x in B:
    for y in B:
        print(toStr(x), 'or', toStr(y), '=', x or y)
```

```
True  or True  = True
True  or False = True
False or True  = True
False or False = False
```

Finally, the negation operator `not` works as expected:

[39]:
```
for x in B:
    print('not', toStr(x), '=', not x)
```

```
not True  = False
not False = True
```

Boolean values are created by comparing numbers using the comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`: * a `==` b is true iff a is equal to b. * a `!=` b is true iff a is different from b. * a `<` b is true iff a is less than b. * a `<=` b is true iff a is less than or equal to b. * a `>=` b is true iff a is bigger than or equal to b. * a `>` b is true iff a is bigger than b.

[40]:
```
1 == 2
```

[40]:
```
False
```

[41]:
```
1 != 2
```

[41]:
```
True
```

[42]:
```
1 < 2
```

[42]:
```
True
```

[43]:
```
1 <= 2
```

[43]:
```
True
```

[44]:
```
1 > 2
```

[44]: False

[45]:
```
1 >= 2
```

[45]: False

Comparison operators can be chained as shown in the following example:

[46]:
```
1 < 2 < 3
```

[46]: True

Python supports the universal quantifier $\forall$. If L is a list of Boolean values, then we can check whether all elements of L are True by writing

$$\texttt{all}(L).$$

For example, to check whether all elements of a list L are even we can write the following:

[47]:
```
L = [2, 4, 6, 7]
all([x % 2 == 0 for x in L])
```

[47]: False

Python also supports the existential quantifier $\exists$. If L is a list of Boolean values, the expression

$$\texttt{any}(L)$$

is true iff there exists an element $x \in L$ such that x is true.

[48]:
```
L = [n for n in range(1, 10)]
any([n ** 2 > 2 ** n for n in L])
```

[48]: True

1.5 Control Structures

First of all, *Python* supports *branching* via the keywords `if`, `elif`, and `else`. The following example is taken from the *Python* tutorial at https://python.org:

[49]:
```
s = input("Please enter an integer: ")
x = int(s)
if x < 0:
    print('The number is negative!')
elif x == 0:
    print('The number is zero.')
elif x == 1:
    print("It's a one.")
```

```
else:
    print("It's more than one.")
```

```
Please enter an integer: 1
It's a one.
```

Loops can be created with the keyword `for`. They can be used to iterate over lists.
The syntax of a `for`-loop is:

```
for x in L:
    S1
    ...
    Sn
```

Here, `x` is the name of a variable that is used to iterate over the elements of the list `L`. The statements `S1`, $\cdots$, `Sn` make up the *body* of the loop. These statements are executed for every element of the list `L`.

The next examples shows a simple loop that prints all the elements from the list `L`:

[50]:
```
for x in L:
    print(x)
```

```
1
2
3
4
5
6
7
8
9
```

Another way to write a *loop* is to use the keyword `while` as in the following example:

[51]:
```
x = 1
while x < 10:
    print(x)
    x += 1
```

```
1
2
3
4
5
6
7
8
9
```

The general syntax of a `while`-loop is as follows:

```
while e:
    S1
    ...
    Sn
```

Here, `while` is the keyword starting the `while`-loop, while `e` is an expression that contains some variable `x` which is presumably updated in the statements `S1`, $\cdots$, `Sn`. These statements are executed as long as the expression `e` is `True`.

The following program computes the prime numbers according to an algorithm given by Eratosthenes. 1. We set n equal to 100 as we want to compute the set all prime numbers less or equal that 100. 2. `primes` is the list of numbers from 0 upto n, i.e. we have initially

$$\texttt{primes} = [0, 1, 2, \cdots, n]$$

Therefore, we have initially

$$\texttt{primes}[i] = i \quad \text{for all } i \in \{0, 1, \cdots, n\}.$$

The idea is to set `primes`$[i]$ to zero iff i is a proper product of two numbers. 3. To this end we iterate over all i and j from the list $[2, \cdots, n]$ and set the product `primes`$[i \cdot j]$ to zero.
This is achieved by the two `for` loops in line 4 and line 5 below.

```
Note that we have to check that the product $i * j$ is not bigger than $n$
for otherwise we would get an *out of range error* when trying to assign $\texttt{primes}[i*j]$.
```

4. After the iteration, all non-prime elements greater than one of the list primes have been set to zero.
5. Finally, we compute the list of all prime numbers by collecting those elements from the list `primes` that have not been set to 0.

[52]:
```
n          = 100
primes     = [i for i in range(0, n+1)]
primes[1] = 0
for i in range(2, n+1):
    for j in range(2, n+1):
        if i * j <= n:
            primes[i * j] = 0
print(primes)
print([i for i in range(2, n+1) if primes[i] != 0])
```

```
[0, 0, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, 0, 0, 0, 17, 0, 19, 0, 0, 0, 23, 0,
0, 0, 0, 0, 29, 0, 31, 0, 0, 0, 0, 0, 37, 0, 0, 0, 41, 0, 43, 0, 0, 0, 47, 0, 0,
0, 0, 0, 53, 0, 0, 0, 0, 0, 59, 0, 61, 0, 0, 0, 0, 0, 67, 0, 0, 0, 71, 0, 73, 0,
0, 0, 0, 0, 79, 0, 0, 0, 83, 0, 0, 0, 0, 0, 89, 0, 0, 0, 0, 0, 0, 0, 97, 0, 0,
0]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]
```

The algorithm given above can be improved by using the following observations: 1. If a number x can be written as a product $a \cdot b$, then at least one of the numbers a or b has to be less than $\sqrt{x}$. Therefore, the `for` loop in line 5 below iterates as long as $i \leq \sqrt{x}$. The function `ceil` is needed

to cast the square root of x to a natural number. In order to use the functions `sqrt` and `ceil` we have to import them from the module `math`. This is done in line 1 of the program shown below. 2. When we iterate over j in the inner loop, it is sufficient if we start with $j = i$ since all products of the form $i \cdot j$ where $j < i$ have already been eliminated at the time when the multiples of i had been eliminated. 3. If `primes`$[i] = 0$, then i is not a prime and hence it has to be a product of two numbers a and b both of which are smaller than i. However, since all the multiples of a and b have already been eliminated, there is no point in eliminating the multiples of i since these are also multiples of both a and b and hence they have already been eliminated.
Therefore, if `primes`$[i] = 0$ we can immediately jump to the next value of i.
This is achieved by the *continue* statement in line 8 below.

The program shown below is easily capable of computing all prime numbers less than a million. On my desktop computer, which is a 2017 iMac with a Quad-Core Intel i5 processor, this takes less than a second.

[53]:
```
%%time
import math

n = 1000000
primes = list(range(n+1))
for i in range(2, math.ceil(math.sqrt(n))):
    if primes[i] == 0:
        continue
    j = i
    while i * j <= n:
        primes[i * j] = 0
        j += 1
P = [i for i in range(2, n+1) if primes[i] != 0]
print(P)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
...
999749, 999763, 999769, 999773, 999809, 999853, 999863, 999883, 999907, 999917,
999931, 999953, 999959, 999961, 999979, 999983]
CPU times: user 728 ms, sys: 13 ms, total: 741 ms
Wall time: 745 ms
```

1.6 Numerical Functions

Python provides all of the mathematical functions that you have learned about at school. A detailed listing of these functions can be found at https://docs.python.org/3.6/library/math.html. We just show the most important functions and constants. In order to make the module `math` available, we use the following `import` statement:

[54]:
```
import math
```

The mathematical constant Pi, which in mathematics is written as π, is available as `math.pi`.

[55]:
```
math.pi
```

[55]:
```
3.141592653589793
```

The sine function is called as follows:

[56]:
```
math.sin(math.pi/6)
```

[56]:
```
0.49999999999999994
```

The cosine function is called as follows:

[57]:
```
math.cos(0.0)
```

[57]:
```
1.0
```

The tangent function is called as follows:

[58]:
```
math.tan(math.pi/4)
```

[58]:
```
0.9999999999999999
```

The *arc sine*, *arc cosine*, and *arc tangent* are called by prefixing the character `'a'` to the name of the function as seen below:

[59]:
```
math.asin(1.0)
```

[59]:
```
1.5707963267948966
```

[60]:
```
math.acos(1.0)
```

[60]:
```
0.0
```

[61]:
```
math.atan(1.0)
```

[61]:
```
0.7853981633974483
```

Euler's number e is stored as the constant `math.e`:

[62]:
```
math.e
```

[62]:
```
2.718281828459045
```

The exponential function $\exp(x) := e^x$ is computed as follows:

[63]:
```
math.exp(1)
```

[63]:
```
2.718281828459045
```

The natural logarithm $\ln(x)$, which is defined as the inverse of the function $\exp(x)$, is called `log` (instead of `ln`):

[64]:
```
math.log(math.e * math.e)
```

[64]:
```
2.0
```

The square root $\sqrt{x}$ of a number x is computed using the function `sqrt`:

[65]:
```
math.sqrt(2)
```

[65]:
```
1.4142135623730951
```

The flooring function `floor`(x) truncates a floating point number x down to the biggest integer number less or equal to x:

$$\texttt{floor}(x) = \max(\{n \in \mathbb{Z} \mid n \leq x\}$$

[66]:
```
math.floor(1.999)
```

[66]:
```
1
```

The ceiling function `ceil`(x) rounds a floating point number x up to the next integer number bigger or equal to x:

$$\texttt{ceil}(x) = \min(\{n \in \mathbb{Z} \mid x \leq n\}$$

[67]:
```
math.ceil(1.001)
```

[67]:
```
2
```

The function `round` rounds its input to the nearest integer. For an integer number n, the floating point number $n.5$ is rounded to $n + 1$.

[68]:
```
round(1.5), round(1.39), round(1.51)
```

[68]:
```
(2, 1, 2)
```

The function `abs` computes the absolute value. Note that this function is not part of the math library but rather is a predefined function. Therefore the name `abs` is not prefixed with the package name `math`.

[69]:
```
abs(-1)
```

[69]:
```
1
```

1.7 The Help System

Typing a single question mark '?' starts the help system of the *Jupyter* notebook.

[70]:
```
?
```

If the name of a module is followed by a question mark, a description of the module is printed.

[71]:
```
math?
```

This also works for functions defined in a module.

[72]:
```
math.sin?
```

The question mark operator only works inside a Jupyter notebook.

Furthermore, there is the function **help**, which is part of *Python*.

[73]:
```
help(math)
```

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.10/library/math.html

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.
```

```
    atan(x, /)
        Return the arc tangent (measured in radians) of x.

        The result is between -pi/2 and pi/2.

        ...

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /opt/anaconda3/envs/algo/lib/python3.10/lib-
dynload/math.cpython-310-darwin.so
```

[74]:
```
help(math.sin)
```

```
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

[75]:
```
help(dir)
```

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

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
    Else, return an alphabetized list of names comprising (some of) the
attributes
    of the given object, and of attributes reachable from it.
    If the object supplies a method named __dir__, it will be used; otherwise
    the default dir() logic is used and returns:
      for a module object: the module's attributes.
      for a class object:  its attributes, and recursively the attributes
        of its bases.
      for any other object: its attributes, its class's attributes, and
        recursively the attributes of its class's base classes.
```

We can use the function `dir()` to print the names of the variables that have been defined.

[76]:
```
dir()
```

[76]:
```
['B',
 'E',
 'HTML',
 'In',
 'L',
 'Out',
 'P',
 '_',
 '_1',
 '_2',
 ...
 '_i',
 '_i1',
 '_i2',
 ...
 'x',
 'y']
```

The names of the form _in where n is a number refer to the *input* of the n^{th} cell of this notebook. Let us check the input of the 55^{th} cell of this notebook.

[77]:
```
_i55
```

[77]:
```
'math.pi'
```

The names of the form _n where n is a number refer to the *output* produced by the n^{th} cell of this notebook.

[78]:
```
_55
```

[78]:
```
3.141592653589793
```

[79]:
```
2 * 3
```

[79]:
```
6
```

Inside a *Jupyter* notebook, the variable `_` refers to the value of the last expression that has been executed.

[80]:
```
_
```

[80]:
```
6
```

The magic command %quickref prints an overview of the so called magic commands that are available in Jupyter notebooks.

[81]:
```
%quickref
```

We can use the command `ls` to list the files in the current directory.

[82]:
```
ls -al
```

```
total 1352
drwxr-xr-x@  4 stroetmann  staff     128 Sep 30 16:10 ./
drwxr-xr-x@ 24 stroetmann  staff     768 Sep 20 13:59 ../
drwxr-xr-x@  3 stroetmann  staff      96 Sep 13 16:58
.ipynb_checkpoints/
-rw-r--r--@  1 stroetmann  staff  691296 Sep 30 16:10 Introduction.ipynb
```

Prefixing a shell command with a `!` executes this command in a shell. On the Windows operating system, the system command to list the files in the current directory is called `dir`, while on Linux and MacOS, the corresponding command is known as `ls`.

[83]:
```
!ls
```

```
Introduction.ipynb
```