Lecture-1

April 12, 2024

0.1 Variables and types

0.1.1 Symbol names

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some special characters such as `_`. Normal variable names must start with a letter.

By convention, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

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

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

0.1.2 Variable Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

[2]:
```
# variable assignments
x = 1.0>Y
my_favorite_variable = 12.2

x
```

[2]:
```
1.0
```

Although not explicitly specified, a variable does have a type associated with it. The type is derived from the value that was assigned to it.

[120]:
```
type(x)
```

[120]:
```
float
```

If we assign a new value to a variable, its type can change.

[121]:
```
x = 1
```

[123]:
```
type(x)
```

[123]:
```
int
```

If we try to use a variable that has not yet been defined we get an `NameError`:

[124]:
```
print(y)
```

```
20
```

0.1.3 Number types

Python allows as any programming language different types of variables. The type of a variable can be always accessed with the help of the *type()* command.

Integers Integers are 32 bit binary numbers and extend from -2^{31} to 2^{31}-1. Python treats numbers without a decimal point automatically as an integer.

[125]:
```
# integer number
x = 1
type(x)
```

[125]:
```
int
```

Floating Point Floating point values are values with a decimal point and go between $\pm 2 \times 10^{-308}$ and $\pm 2 \times 10^{308}$

[126]:
```
# float variable
x= 3.141
```

Complex Numbers

[22]:
```
c=2+4j
type(c)
```

[22]:
```
complex
```

Complex numbers have built in *accessors*. These accessors give for example access to the *real* and *imaginary* part of the complex number.

[23]:
```
r=c.real
print(r)
```

```
2.0
```

[24]:
```
i=c.imag
print(i)
```

```
4.0
```

On the other side, one my also evaluate the complex conjugate of a complex number by one of those accessors. Note that this is provided by a function here, while the above real and imaginary part are values. The are some basic functions available, which act on complex numbers. More complex calculations are possible with functions built in to modules such as *cmath* or *numpy*.

[25]:
```
c=(2+4j).conjugate()
print(c.imag)
```

```
-4.0
```

Type casting

[56]:
```
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

[57]:
```
x = int(x)

print(x, type(x))
```

```
1 <class 'int'>
```

[58]:
```
z = complex(x)

print(z, type(z))
```

```
(1+0j) <class 'complex'>
```

[59]:
```
x = float(z)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-59-19c840f40bd8> in <module>
----> 1 x = float(z)

TypeError: can't convert complex to float
```

Complex variables cannot be cast to floats or integers. We need to use `z.real` or `z.imag` to extract the part of the complex number we want:

[60]:
```
y = bool(z.real)

print(z.real, " -> ", y, type(y))

y = bool(z.imag)
```

```
print(z.imag, " -> ", y, type(y))
```

```
1.0  ->  True <class 'bool'>
0.0  ->  False <class 'bool'>
```

0.2 Operators and comparisons

Most operators and comparisons in Python work as one would expect:

- Arithmetic operators `+`, `-`, `*`, `/`, `//` (integer division), '**' power

[28]:
```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

[28]:
```
(3, -1, 2, 0.5)
```

[29]:
```
1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

[29]:
```
(3.0, -1.0, 2.0, 0.5)
```

[30]:
```
# Integer division of float numbers
3.0 // 2.0
```

[30]:
```
1.0
```

[31]:
```
# Note! The power operators in python isn't ^, but **
2 ** 2
```

[31]:
```
4
```

Note: The `/` operator always performs a floating point division in Python 3.x. This is not true in Python 2.x, where the result of `/` is always an integer if the operands are integers. to be more specific, `1/2 = 0.5` (`float`) in Python 3.x, and `1/2 = 0` (`int`) in Python 2.x (but `1.0/2 = 0.5` in Python 2.x).

- The boolean operators are spelled out as the words `and`, `not`, `or`.

[32]:
```
True and False
```

[32]:
```
False
```

[33]:
```
not False
```

[33]:
```
True
```

[34]:
```
True or False
```

[34]:
```
True
```

- Comparison operators `>`, `<`, `>=` (greater or equal), `<=` (less or equal), `==` equality, `is` identical.

[35]:
```
2 > 1, 2 < 1
```

[35]:
```
(True, False)
```

[36]:
```
2 > 2, 2 < 2
```

[36]:
```
(False, False)
```

[37]:
```
2 >= 2, 2 <= 2
```

[37]:
```
(True, True)
```

[38]:
```
# equality
[1,2] == [1,2]
```

[38]:
```
True
```

[39]:
```
# objects identical?
l1 = l2 = [1,2]

l1 is l2
```

[39]:
```
True
```

0.3 Data Types in Python

Now that we know a bit about the general use of Python, it's time to look at different data types we may find useful in our course. Besides the data types for the different number types mentioned above, there are also other types like **strings**, **lists**, **tuples** and **dictionaries**. There are usually a number of *methods* (functions) connected to each data type providing useful functions. We will not talk about these methods, except we need them. It is up to you to explore the internet a bit and search for suitable methods for your purpose.

The follwing few cells will give you a short introduction into each type.

0.3.1 Strings

Strings are lists of keyboard characters as well as other characters not on your keyboard. They are not particularly interesting in scientific computing, but they are nevertheless necessary and useful. Texts on programming with Python typically devote a good deal of time and space to learning about strings and how to manipulate them. Our uses of them are rather modest, however, so we take a minimalist's approach and only introduce a few of their features.

[61]:
```
s='Hello' # string variable
```

[62]:
```
t="World!"
```

String can be concatenated using the + operator.

[60]:
```
c=s+' '+t
```

[61]:
```
print(c)
```

```
Hello World!
```

As strings are lists, each character in a string can be accessed by addressing the position in the string (see Lists section)

[62]:
```
c[1]
```

[62]:
```
'e'
```

Strings can also be made out of numbers.

[63]:
```
"975"+"321"
```

[63]:
```
'975321'
```

If you want to obtain a number of a string, you can use what is known as type casting. Using type casting you may convert the string or any other data type into a different type if this is possible. To find out if a string is a pure number you may use the str.isnumeric() method. For the above string, we may want to do a conversion to the type *int* by typing:

[64]:
```
("975"+"321").isnumeric() # or you may use as well str.isnumeric("975"+"321")
```

[64]:
```
True
```

[65]:
```
int("975"+"321")
```

[65]:
```
975321
```

There are a number of methods connected to the string data type. Usually the relate to formatting or finding sub-strings. Formatting will be a topic in our next lecture. Here we just refer to one simple find example.

[66]:
```
t.find('ld') ## returns the index at which the sub string 'ld' starts in t
```

[66]:
```
3
```

[67]:
```
t.capitalize()
```

[67]:
```
'World!'
```

0.3.2 Lists

Lists are another data structure, similar to NumPy arrays, but unlike NumPy arrays, lists are a part of core Python. Lists have a variety of uses. They are useful, for example, in various bookkeeping tasks that arise in computer programming. Like arrays, they are sometimes used to

store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. So in general, we prefer arrays to lists for working with scientific data. For other tasks, lists work just fine and can even be preferable to arrays.

[105]:
```
a = [0, 1, 1, 2, 3, 5, 8, 13]
```

[91]:
```
b = [5., "girl", 2+0j, "horse", 21]
```

The length of a list can be obtained by the *len()* command if you need the number of elements in the list for your calculations.

[92]:
```
len(b)
```

[92]:
```
5
```

There are powerful ways to iterate through a list and also through arrays in form of *iterator*. We will talk about them later in more detail. Here is an example, which shows the powerful options you have in Python. The example will go through the list **b** and return all elements of the type str.

[93]:
```
[element for element in b if type(element)==str]
```

[93]:
```
['girl', 'horse']
```

Individual elements in a list can be accessed by the variable name and the number (index) of the list element put in square brackets. Note that the index for the elements start at *0* for the first element (left).

[94]:
```
b[1]
```

[94]:
```
'girl'
```

Elements may be also accessed from the back by nagative indices. *b[-1]* denotes the last element in the list and *b[-2]*, the element before the last.

[95]:
```
b[-1]
```

[95]:
```
21
```

[96]:
```
b[-2]
```

[96]:
```
'horse'
```

Individual elements in a list can be replaced by assigning a new value to them

[97]:
```
b[-2]='cat'
```

[98]:
```
b
```

[98]:
```
[5.0, 'girl', (2+0j), 'cat', 21]
```

Lists can be concatanated by the + operator

[106]:
```
c=a+b
c
```

[106]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 5.0, 'girl', (2+0j), 'cat', 21]
```

A very useful feature for lists in python is the **slicing** of lists. Slicing means, that we access only a range of elements in the list, i.e. element 3 to 7. This is done by inserting the starting and the ending element number separated by a colon (:) in the square brackets. The index numbers can be positive or negative again.

[100]:
```
c[3:7]
```

[100]:
```
[2, 3, 5, 8]
```

Inserting a second colon behind the ending element index together with a thrid number allows even to select only ever second or third element from a list.

[101]:
```
c[3:9:2]
```

[101]:
```
[2, 5, 13]
```

It is sometimes also useful to reverse a list. This can be easily done with the reverse command.

[103]:
```
c.reverse()
c
```

[103]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 5.0, 'girl', (2+0j), 'cat', 21]
```

Lists may be created in different ways. An empty list can be created by assigning emtpy square brackets to a variable name. You can append elements to the list with the help of the append command which has to be added to the variable name as shown below. This way of adding a particular function, which is part of a certain variable class is part of object oriented programming.

[87]:
```
a=[]
```

[88]:
```
a.append('h')
```

A list of numbers can be easily created by the *range()* command.

[76]:
```
range(10)
```

[76]:
```
range(0, 10)
```

[77]:
```
range(3,10)
```

[77]:
```
range(3, 10)
```

[78]:
```
range(3,10,2)
```

[78]:
```
range(3, 10, 2)
```

Lists (and also tuples below) can be multidimensional as well, i.e. for an image. The individual elements may then be addressed by supplying two indices in two square brackets.

[79]:
```
a = [[3, 9], [8, 5], [11, 1]]
```

[80]:
```
a[1]
```

[80]:
```
[8, 5]
```

[81]:
```
a[1][0]
```

[81]:
```
8
```

0.3.3 Tuples

Tuples are also list, but immutable. That means, if a tuple has been once defined, it cannot be changed. Try to change an element to see the result.

[64]:
```
point = (10, 20)

print(point, type(point))
```

```
(10, 20) <class 'tuple'>
```

Tuples may be unpacked, e.g. its values may be assigned to normal variables in the following way

[65]:
```
x, y = point

print("x =", x)
print("y =", y)
```

```
x = 10
y = 20
```

0.3.4 Dictionaries

Dictionaries are like lists, but the elements of dictionaries are accessed in a different way than for lists. The elements of lists and arrays are numbered consecutively, and to access an element of a list or an array, you simply refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by "keys", which can be either strings or (arbitrary) integers (in no particular order). Dictionaries are an important part of core Python. However, we do not make much use of them in this introduction to scientific Python, so our discussion of them is limited.

[66]:
```
room = {"Ralf":422, "Frank":322, "Dekan":550}
```

[67]:
```
room['Ralf']
```

[67]:
```
422
```

[68]:
```
room.keys()
```

[68]:
```
dict_keys(['Ralf', 'Frank', 'Dekan'])
```

[69]:
```
room.values()
```

[69]:
```
dict_values([422, 322, 550])
```

0.4 Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more.

To use a module in a Python program it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

[1]:
```
import math

x = math.cos(2 * math.pi)

print(x)
```

```
1.0
```

This includes the whole module and makes it available for use later in the program. Alternatively, we can chose to import all symbols (functions and variables) in a module to the current namespace (so that we don't need to use the prefix "`math.`" every time we use something from the `math` module:

[2]:
```
from math import *

x = cos(2 * pi)

print(x)
```

```
1.0
```

This pattern can be very convenient, but in large programs that include many modules it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would elminate potentially confusing problems with name space collisions.

0.4.1 Contents and documentation of a module

Once a module is imported, we can list the symbols it provides using the `dir` function:

[127]:
```
import math

print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

[128]:
```
help(math.log)
```

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

log(…)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

[130]:
```
math.log(10)
```

[130]:
```
2.302585092994046
```

[131]:
```
math.log(10, 2)
```

[131]:
```
3.3219280948873626
```

We can also use the `help` function directly on modules: Try

`help(math)`

Some very useful modules form the Python standard library are `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

A complete lists of standard modules for Python 3 is available at http://docs.python.org/3/library/ .

0.5 Exceptions

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and fallback to somewhere else in the code where the closest try-except statement is defined.

To generate an exception we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseException` or a class derived from it.

[13]:
```
raise Exception("description of the error")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-13-c32f93e4dfa0> in <module>
----> 1 raise Exception("description of the error")

Exception: description of the error
```

A typical use of exceptions is to abort functions when some error condition occurs, for example:

```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # rest of the code goes here
```

To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the `try` and `except` statements:

```
try:
    # normal code goes here
except:
    # code for error handling goes here
    # this code is not executed unless the code
    # above generated an error
```

For example:

[14]:
```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except:
    print("Caught an exception")
```

```
test
Caught an exception
```

To get information about the error, we can access the `Exception` class instance that describes the exception by using for example:

```
except Exception as e:
```

[15]:
```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except Exception as e:
    print("Caught an exception:" + str(e))
```

```
test
Caught an exception:name 'test' is not defined
```

0.6 What's next

This has been a short overview over the basic things in Jupyter and Python. We will address more details on the way to our physical problems and data analysis code. As one of the most fundamental features required in physics is plotting of data, we will address plotting with a first primer next week in Lecture 2. Also input and output including keybord and files will be of our concern.