Python Cheat Sheet

by Patrick Grössing

Python code format

- In contrast to C the scope of a function or block of code is not defined via curly braces. To differentiate between statement blocks python uses indentation of code. The next part shows an example of simple python code with indentation. Do not get confused by the last `if` statement. It will be explained in the section `Variable assignment`.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def main():

    answer = "yes"
    if (answer == "yes"):
        print( "I am a robot and shall not proceed." )
    elif ( answer == "no" ):
        print( "Hello World!" )
    else:
        print( "You did not know what to say did you?" )

if __name__ == "__main__":
    main()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Remark: It is important that one keeps a consistent intendation througout the sourcecode. Therefore always use tabs with consistent spacing (e.g. 4 whitespace characters for one tab is reasonable). NEVER start mixing whitespaces and tabs for indentation.

Base Types & Container Types

- int, float, bool, str, bytes (No specific declaration of type needed. Interpreter assigns the right type)
- list (Ordered sequence of modifiable values)
  - e.g. `[1, 2, 3, 4]`

- tuple (Ordered sequence of non-modifiable (immutable) values)
  - e.g. `(1, 2, 3, 4)`
- dict (Set of key and value pairs. Values can be accessed by keys)
  - e.g. `{1:"one", 3:"three", "key":"value"}`

Remark: In this course we will mostly try to avoid those standard containers. Numpy arrays offer better performance, memory footprint and flexibility when it comes to number crunching. In some cases it can be beneficial to use lists or tuples, e.g., appending in a for loop.

Variable assignment

- Since declaration of the variable base type is handled by the interpreter assignment is straightforward:
  - `a = 10` (`a` is now an integer with the value 10)
  - `intlist = [1, 2, 3, 4]` (`intlist` is now a list)
  - `a=b=c=0`
  - `a, b = 9, 10` (Multiple assignments in one line are also possible and also often used)
  - `a=None` (Assign undefined constant value to a)
- Python is case sensitive in variable names
- Language keywords are forbidden (e.g. list) as variable names. Also try to avoid leading and ending underscores (e.g. `__init__` or `__name__`). This syntax is reserved for special class methods.
- The interpreter sets internal variables when used (e.g. `__debug__`). One of this variables is called `__name__`. When interpreting a python script this variable is set to the string `__main__`. This triggers the if statement and executes the main function of the script.

Conversion & Incrementation

- As in C it is possible to cast or convert numbers or variables into different base types.
  - `int("10")` ⇒ 10 (Conversion from string to int)
  - `int(10.11)` ⇒ 10 (Truncation of float. This is different to rounding!)
  - `list("foo")` ⇒ `['f', 'o', 'o']`
  - `bool(a)` (False for a = 0, empty a, None or False, True otherwise)
- Numbers can also be incremented by a simple syntax:
  - `x+=3` ⇔ `x=x+3`

List indexing, slicing and operations on lists

- One can access elements in a list directly by indices, e.g., the list `bar = [1, 2, 3, 4, 5, 6]`
  - `bar[0]` ⇒ 1 (Indices start at 0 in Python)
  - `bar[-1]` ⇒ 6 (This accesses the last element. Very useful if one does not know the length of the list)
- Slicing returns a reduced representation of a list without using explicit loops (`bar[start:stop:step]` start and stop are indices).
  - `bar[::2]` ⇒ `[1, 3, 5]` (Returns every second element)
  - `bar[1:5:2]` ⇒ `[2, 4]` (Value at start index is returned whereas value at stop index is omitted)
  - `bar[::-2]` ⇒ `[6, 5, 2]` (With minus in step one can reverse the list)
- Operations on lists with numpy equivalents
  - `len(bar)` ⇒ 6 (Gives the length of the list. Equiv: numpy.size)
  - `max(bar)` ⇒ 6 (Maximum value. Equiv: numpy.max)
  - `4 in bar` ⇒ True (Searches for variable in list and returns True if found)
  - `enumerate(bar)` (Returns an iterator of the list and can be used to give list indices in for loops. Equiv: numpy.ndenumerate)
  - `bar.append(7)` ⇒ `[1, 2, 3, 4, 5, 6, 7]` (Append the given element to given list. Useful in for loops.)
  - `bar.sort()`, `bar.reverse()` etc.

Remark: All the mentioned functions and methods also work on numpy arrays (Except the append method). When handling Numpy arrays one should always resort to the Numpy equivalents because of better performance and a greater amount of options. For example when dealing with a 2-dim Numpy array (a matrix) `numpy.size` can return the total amount of items, the number of columns or the number of rows whereas `len()` always returns the number of rows.

Conditional statements & Boolean logic

- Like in every programming language one can compare numbers and do boolean logic:
  - `<`, `>`, `<=`, `>=`, `==`, `!=`
  - `a and b`, `a or b` (This works for a and b being booleans. Remember that the `and` operator has precedence)
  - It is not sensible to compare floats with `!=`, `==` as the comparison can yield unintended results basing off numerical errors. We will see special numpy routines dealing with this problem (`numpy.isclose`, `numpy.allclose`)

- We can achieve control flow inside the code with the help of the `if` statement. For the syntax look at the first example in this cheat sheet.
  - Conditions inside the if statement are evaluated from left to right.
  - As everwhere in python the intendation is important. The colon is at the start of every loop, function definition or control flow statement.

Iterative and conditional loops

- The conditional loop is called the while loop. It is executed as long as the logical condition evaluates to `True`

```
i = 0

while (i < 10):
    print('The count is:', i)
    i+=1
```

⇒ "The count is 0"; "The count is 1"; …

- The for loop works a bit different to the for loops known from C. The for loop takes an arbitrary iterator or generator as input and loops those values.
  - In python one rarely accesses list or matrix values with their index (This is considered 'unpythonic' code). To access list elements a simple syntax is used:

```
lst = [1, 2, 3, 4, 5, 6, 7, 8]
for x in lst:
    print("The current value is: ", x)
```

⇒ "The current value is 1"; "The current value is 2"; …

- If really necessary one can also access the indices by iterating the length of the list:
- The range operator generates an iterator from a number. In this case the length of the list is used.

```
lst = [1, 2, 3, 4, 5, 6, 7, 8]
for i in range(len(lst)):
    print("The current value is: ", lst[i])
```

⇒ "The current value is 1"; "The current value is 2"; …

- It is also possible to access index and variable at the same time. For this one needs the `enumerate` operator.

```
lst = [1, 2, 3, 4, 5, 6, 7, 8]
for i, x in enumerate(lst):
    print("The currend index is: ", i)
    print(" The current value is: ", x)
```

⇒ “The current index is: 0”; “The current value is: 1”; . . .

- Iterating over numpy arrays can be done in a similiar fashion. For better flexibility and efficiency one should use the `numpy.nditer` iterator

Remark: Since numpy offers the ability to express array operations without explicitly calling loops, `for` or `while` loops should be a rare sight in your code. This also helps to keep up the performance needed in numerical calculations.

Prints and assert

- In our code we will mostly use `print` and `assert` for debugging purposes
  - `print("Hello World!", var, "This is a variable")` (Standard simple print command including a variable `var`)
  - `print("%1.2f This is a float. %i This is a integer" %(2.333, 10))` ⇒ “2.33 This is a float. 10 This is a integer” (Formatted printing like with `printf` in C-Code)
- For our purposes the assert statement can be seen as an if condition combined with a print function. As long as the condition evaluates to true nothing happens. When it evaluates to False an error is thrown and the assertion argument printed with it.
  - `assert (Temperature >= 0), "Temperature can not be negative!"`
- Asserts are mainly used to make sanity checks on the code. In contrast to print statements they can be turned off in the code by providing the `-O` flag to the interpreter (This sets the `__debug__` internal variable to `False`).

Loading modules

- For our numerical purposes we are mostly relying on special libraries speeding up our calculations as well as our programming work tremendously. These special libraries need to be imported in order to work as they are not part of the standard python library.
- One can import the whole library or only parts of it. The syntax is the following:
  - `import numpy as np` (This includes all of the numpy library under the alias `np`). If one would now try to define an numpy array the syntax is: `np.array([1, 2, 3])`
  - This declaration has to be done at the top of the python script before function definitions
  - It is often useful to import needed parts of the library with a different name to get clear and concise code. E.g. one could call the numpy linalg routine `eigh` with `numpy.linalg.eigh` or use `from numpy import linalg as LA` and then `LA.eigh` instead.
  - To import all methods from a certain part of the library one could also use e.g. `from numpy.linalg import *` now the diagonalisation routines could just be called with e.g. `eigh`

Remark: Numpy and Scipy internally call Fortran routines from the `LAPACK` and `BLAS` libraries. This is the reason the code is much faster and more efficient than native Python code. This is also the reason we will mostly concentrate on internal numpy containers (`numpy.array`) and try to avoid long python loops.

Defining functions

- The syntax for defining a function looks like this (The arguments are just a representation of all the possibilities):

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def foo( a, b ,c , *args, **kwargs):
    print( args )
    for key in kwargs:
        print(key, kwargs[key])
    return a, b, c
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- A function does not need to be declared before definition!
- A function can also return multiple values or containers. If the function returns multiple values the syntax for calling it looks like this:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
var1, var2, var3 = foo(a, b, c, bar1, bar2, key_one="Hello World!")
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- `*args` (arguments) makes it possible to pass a flexible number of parameters to a function. In this case bar1 and bar2 would be wrapped up in a tuple and can be accessed seperately via `args[index]` inside the function.
- `**kwargs` (keyword arguments) takes keyworded variables and can be accessed by the key given in the definition (`key_one`). The code above would print `key_one Hello World` in the example above.
- You will often see keyworded arguments in the definition of numpy, scipy and matplotlib functions. For example: `numpy.concatenate((a, b), axis=0)` here `axis` is the key and `0` the argument.