Programming in Python

by Youjun Hu

Institute of Plasma Physics, Chinese Academy of Sciences
Email: yjhu@ipp.cas.cn

1 Introduction

Python is getting so popular that nearly every scientist needs to know a little bit of Python, in a similar way that a scientist needs to know English. Python is generally considered as a diverse and intuitive language, which is suitable for both beginners and experienced programmers. Many high school students in USA are familar with python and many famous companies like Google and Facebook are using Python. A popular language is worth learning. Python can be used:

- alongside other softwares to create workflows (system scripting), as a "glue" language. Many people use python to avoid writing shell scripts in Linux, considering the irregular syntax of shell script languages.

- to handle big data and perform complex mathematics (number crunching, visulaztion).

My primary use of Python is to read numerical data, analyze and visualize these data by using some libraries such as `numpy` and `matplotlib`.

- All Python (intepretor) releases are open source. Python has lots of libraries, documentation, and an active community.

- Python works on different platforms (Linux, Windows, Mac, etc). Python runs in interpreting mode, meaning that code can be executed as soon as it is written.

- Python was designed for convenience, and uses **dynamically typed** variables.

- Python uses lexical scope, like most contemporary programming languages.

- Python can be treated in a **procedural** way, an **object-orientated** way or a **functional** way.

One of Python zens is "Readability counts". To implement this, Python relies on indentation, using whitespace, to define blocks, such as loops, branchings, function and class definitions. This enhances source code readability. Other programming languages often use brackets for this purpose (e.g., `C` and `Java` use curly-brackets). Partially due to this easy-to-eyes syntax (using indentation as level indicators), python codes are considered (by most programmers) as concise, clear, easy-to-learn, and easy-to-read. Writing Python code is very similar to writing pseudo code and this often allows developers to write programs with fewer lines than some other programming languages would allow.

A python statement/expression can be terminated by a newline or semicolon (similar to `Fortran`, different from `C` where newlines are equivalent to whilespces and semicolons at the end of a command is necessary), semicolons at the end of a command is not necessary, but if used, can suppress some output in interactive modes.

Comments in python: (1) anything after a hash (`#`) in a line is a comment. (2) triple double-quotes introduce string literals, which can have multiple-lines, and thus can serve as multiple-line comments.

Python is case sensitive.

Before going to details, it is helpful to have a look at all the keywords in Python (so that we can easily distinguish user variables from these keywords in a syntax form):

Value Keywords: `True, False, None`

Operator Keywords: `and, or, not, in, is`

Control Flow Keywords: `if, elif, else`

Iteration Keywords: `for, while, break, continue, else`

Structure Keywords: `def, class, with, as, pass, lambda`

Returning Keywords: `return, yield`

Import Keywords: `import, from, as`

Exception-Handling Keywords: `try, except, raise, finally, else, assert`

Asynchronous Programming Keywords: `async, await`

Variable Handling Keywords: `del, global, nonlocal`

Two keywords have additional uses beyond their initial use cases. The `else` keyword is also used with loops as well as with try and except. The `as` keyword is also used with the `with` keyword.

1.1 Run Python

1.1.1 Interactive mode

In a Linux terminal, invoking `python` without any comandline option will enable the interactive mode, where we can type python source code. I often use this mode to do simple calculations:

```
yj@pic:~$ python
>>> 2 + 3
5
```

1.1.2 Script file mode

Create a Python script file `a.py` (by using any text editor):

```
x = "World"
print("Hello " + x)
```

Then run it in a Linux terminal by specifying the interpreter and the script file:

```
python a.py
```

Another way (prefered) is to specify the interpreter in the script file, such as:

```
#!/usr/bin/python3
x = "World"
print("Hello " + x)
```

where the first line specifies the interpreter for the present script file. Then we give executable permission to this file:

```
chmod u+x a.py
```

And run it in a Linux terminal (assume that the file is in the present directory):

```
./a.py
```

nomenclature: script vs. module

A plain text file containing Python code that is intended to be directly executed by the user is usually called script, which is an informal term that means top-level program file.

On the other hand, a plain text file, which contains Python code that is designed to be imported and used from another Python file, is called module.

So, the main difference between a module and a script is that modules are meant to be imported, while scripts are made to be directly executed.

1.2 Statement Vs. function call

Arguments in a function call are enclosed in round-brackets whereas arguments to a keyword statement are usually provided without round-brackets. For example:

```
x=1
del x
```

where `del` is a keyword statement and thus its argument `x` is not enclosed by round-brackets. In Python3, `print` becomes a function (in python2, it is a keyword statement). Therefore, arguments to python3 print must be enclosed by round-brackets:

```
print("hello")
```

In Python3, `exec` is a build-in function (rather than a keyword) and thus must be called with parentheses:

```
exec("x=12*7")
```

1.3 Use python library

Use `import` to get access to python libraries. For example:

```
import numpy as np
import matplotlib.pyplot as plt
```

2 Assignments

`=` is used as the assignment operator. In a dynamically typed language as python, since there is no type declariation (which defines a new variable in statically typed languages), the assigenment operation is the most used way of introducing a new variable.

Python assignment statements do not return values. Python supports structured assignments. For example:

```
a,b,c=1,2,3
```

```
(a, (b,c))=(10, (20,30))
```

2.1 Value Type

Python is a language with dynamically typed variables, which means the type of a variable can change during runtime and hence there is no type declaration for variables. More accurately, typing is associated with the value that a variable assumes rather than the variable itself (a variable is a pointer pointing to that value, or more accurately, a variable is a point to the location where a value is stored). A variable is defined/created (i.e., its type is determined and memory is allocated) when we assign a value to a name (note that although the type declaration for variables is not needed, a variable still needs to be defined/ created before we can reference/use it.). For example, `abc = ‘‘hello’’` defines a variable `abc` pointing to a string type (we can use `print(type(abc))` to check the type of the variable.) Complex numbers are written with a `j` as the imaginary part. For example:

```
x = 3+5j
print(type(x))
<class 'complex'>
```

Python's primitive types includes string, int, float, complex, bool. Besides primitive types, python provides many useful data structures, such as lists, sets, dicts (discussed later). For example `abc=[]` defines a variable `abc` pointing to a list type.

2.2 Scope of variables

Python adopts the lexical scope. Only `def` and `lambda` can introduce new scopes. A variable created outside of a function is known as a global variable. A variable created inside a function's body is known as a local variable. Formal arguments identifiers also behave as local variables.

The scope of a variable becomes a little subtle for dynamically typed languages, compared with statically typed languages. In Python, variables that are **only referenced** inside a function are implicitly **global**. If a variable is **assigned** a value anywhere within the function's body, it's assumed to be a **local** unless explicitly declared as `global`.

Therefore, to access a global variable inside a function, the safe way is to declare it as global using `global` key word. If not using `global` keyword, a variable will be considered as a local variable if there is an assignment statement to the variable somewhere in the body. This local variable will shadow the corresponding global variable. The locality of the variable will apply before the assignment creating the local variable, and thus if we use it before the assignment, we will get the UnboundLocalError: local variable 'x" referenced before assignment.

Similarly, in a nested scope, we can use keyword `nonlocal` to declare that a variable is from enclosing scopes (scopes enclosing the present scope).

The LEGB rule is a kind of name lookup procedure, which determines the order in which Python looks up names. If you reference a given name, then Python will look that name up sequentially in the local, enclosing, global, and built-in scope. If the name exits, then you'll get the first occurrence of it. Otherwise, you'll get an error.

3 Define and call a function

Use keyword `def` to define a function. The syntax is as follows:

```
def func_name ( arg1, arg2, ... ):
    statements
    return something
```

The function name follows the keyword `def`. Then the list of parameters, a colon, and a newline follow. Increase indent, the body of the function definition starts (use `return` to return value). Going back to the original indent terminates the function definition.

For example:

```
def sum(ls):
    s=0
    for x in ls:
        s=s+x
    return s
```

Call a function:

```
sum([1,2,3]) # call the function defined above, the result is 6
a = sum([1,2,3]) # call the function and assign the returned value to a variable
```

Python passes sequences by reference and others by value.

You can define another function inside the body of a function, i.e., a nested function.

optional parameters,   parameter default values.

3.1 Nested function and return function as value

When you handle a nested function as data, the statements that make up that function are packaged together with the environment in which they execute. The resulting object is known as a closure. In other words, a closure is an inner or nested function that carries information about its enclosing scope, even though this scope has completed its execution.

In Python, we can define a function inside the body of a function definition (function inside function). And we can return the inner function as the return value of the enclosing function. For example

```
>>> def power_factory(exp):
...     def power(base):
...         return base ** exp
...     return power
...
>>> square = power_factory(2)
>>> square(10)
100
>>> cube = power_factory(3)
>>> cube(10)
1000
```

Variables like `exp` are called free variables. They are variables that are used in a code block but not defined there. Free variables are the mechanism that closures use to retain state information between calls.

4 Built-in functions and library functions

Complete list of python built-in functions can be found at https://docs.python.org/3/library/functions.html

As an example of useful built-in functions, `dir()`, without arguments, return the list of names in the current local scope. With an argument, `dir` attempt to return a list of attributes and methods for that object.

Unlike `Fortran`, mathematical functions like `sin` and `cos` are not built-in functions of Python interpreter. These functions are defined in `math` module. To use them:

```
import math
math.sin(1.0)
```

But, I prefer to use mathematical functions defined in `numpy` module, which are usually more powerful than their counterparts in `math` module. For example:

```
In [5]: import numpy as np
In [6]: np.sin(1.0)
Out[6]: 0.8414709848078965
In [7]: a = [1,2,3] #define a list
In [8]: np.sin(a)
Out[8]: array([0.84147098, 0.90929743, 0.14112001])
```

```
In [9]: math.sin(a)
TypeError: must be real number, not list
```

5 Flow control

5.1 Logical expressions

Python supports the following relation operators:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Belongs to: `a in b`

The value of the above expression is of logical(bool) type. Python's logical operators are keywords `and` and `or`, which can be used to combine logical expressions to form a new logical expression. These logical expressions can be used in several ways, most commonly in `if` structures and `while` loops.

5.2 Conditionals

Python `if` structure takes the following form:

```
if a==0:
    print("a is zero")
elif a<0:
    print("a is negative")
else:
    print("a is positive")
```

The blocks in a `if` structur does not introduce a new scope. Therefore, if new variables are defined in the blocks, they are still visible outside the `if` structure.

One-line `if` structure:

```
if a > b: print("a is greater than b")
```

where we have only one statement to execute and we put it on the same line as the `if` statement.

Short hand `if ... else`. If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
print("A") if a > b else print("B")
```

The advantage of one-line `if` structure is that we can pass the return value seamlessly to other operation (functional style). For example:

```
abs_A = (A if a>=0 else -A)
```

5.3 Loop structure

Python has two primitive loops: `while` loop and `for` loop.

```
i = 1 # while-loop requires relevant variables to be ready
while i < 6:
     print(i)
     i =i + 1
```

The above `while` loop is quite similar to that in Fortran, which looks like the following:

```
i = 1
do while(i<6)
   print *, i
   i = i + 1
enddo
```

Another python loop structure is the `for` loop, which is used for iterating over a **sequence** (that is either a list, a tuple, a dictionary, a set, or a string)

```
for x in sequence:
    body
```

The above `for` loop is less like the `for` loop in C programming language, which looks like the following:

```
for(i=start; i<some_threthod; some_operation_modifying_i) {loop_body}
```

With `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc. For example:

```
fruits = ["apple", "banana", "cherry"]
for  x in fruits:
    print(x)

for x in range(0, 3):
    print(x)
```

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

`continue` and `break` are similar to `cycle` and `exit` of Fortran and can be used in both `while` and `for` structures.

In the above, `x` acts as a dummy variable, which does not need to be defined before the loop. The `for` loop does not form a new scope. This means that the variable `x` is acessible outside the loop. On exist from the `for` loop, the value of `x` is equal to the value got from the last iteration. If `x` is defined before the `for` loop, its value will be modified by the `for` loop.

Note that the keyword `in` is used in Python for two different purposes: (1) The `in` keyword is used to check if a value is present in a sequence (list, range, string etc.). (2) The `in` keyword is also used to iterate through a sequence in a `for` loop.

6 Built-in data structure

Python has four primitive data structures, namely list, tuple, set, and dict:

```
L = [3, "hello", 0.5] # list
```

```
t = (1,2,"apple",3) # tuple
s = {"apple", "banana", "cherry"} #set
d = {"a":1, "b":2} #dict, each element has two filds, key and value, separated by
:
```

Examining the above codes, we can summarize the sytax for differnt data structure: (1) elements in the four data structures are all separated by commas; (2) lists are written with **square-brackets**, tuples are written with **round-brackets** (parentheses), sets and dicts are written with **curly-brackets**; (3) a dict (or hash) is a special set in which each element has two fields, key and value, which are separated by a colon.

Lists and tuples are **ordered** collections and thus support **indexing** whereas sets and dicts are **unordered** and do not support indexing. The difference between a list and a tuple lies in that a list can be modfied but a tuple is unchangable.

The primitive type `string` can also be considered a nontrivial data structure, which supports indexing, similar to a list.

6.1 List

One of the most fundamental data structures in any language is the array. Python does not have a native array data structure, but it has the list data structure which is more general: a list in Python can contain items of various types. For example:

```
myList = [3, "hello", 0.5] #define a list containing three items of different
types
```

Since a list contains misc items, which are not just numbers, some operations on lists are different from array operations that we expect in an array Language. For example:

```
In [12]: a=[1,2,3]
In [13]: 2*a
Out[13]: [1, 2, 3, 1, 2, 3] #rather than doubling each element vale in the list
```

Adding lists concatenates them, just as the "+" operator concatenates strings. To use the standard array operation, we can use `numpy.asarray` to convert a list to an array:

```
In [16]: b=np.asarray(a)
In [17]: 2*b
Out[17]: array([2, 4, 6])
```

The Python standard library defines an array type, which is still a list type, except that the type of objects stored in it is constrained to a single type. The methods of this array type are different from the usual array operation. For example, `2*a` is not to double the value of the each element in the array:

```
In [1]: import array
In [2]: a=array.array('d', [1,2])
In [3]: 2*a
Out[3]: array('d', [1.0, 2.0, 1.0, 2.0])
```

As a result, this array type is not as versatile, efficient, or useful as the NumPy array. We will not be using Python arrays at all. Therefore, whenever we refer to an "array," we mean a "NumPy array."

```
In [1]: import numpy as np
In [2]: a=[1,2,3]
In [3]: b=np.array(a)
In [4]: b
Out[4]: array([1, 2, 3])
In [5]: 2*b
Out[5]: array([2, 4, 6])
```

6.1.1 Addressing and Slicing lists

List elements can be accessed by using indexes. Indexes of a list start from zero, i.e., `myList[0]` corresponds to the first item in the list. Elements of a list can also be accessed by using negative index. For example `myList[-1]` refers to the last element of the list, and `myList[-2]` refers to the next-to-last element of the list, etc.

We can use slicing notation to pick out a sublist, e.g., `b = myList[0:2]`. Python use the convention that the final element specified, i.e. `myList[2]` in this case, is not included in a list slice. If the upper and/or lower limit are omitted, the corresponding list limit will be used, e.g., `myList[:]` refers to the whole list.

Nested lists (multidimensional lists, lists of lists) can be referenced by using multiply index, such as `myList[0][1]`, not `myList[0,1]`. The latter notation only works for `numpy` array objects.

6.1.2 List methods

Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types. A list object has some predefined methods. These methods are invoked in the same way as in other object-orientated Languages, i.e., `instant.method(arg1,arg2,...)`. For example:

```
mylist.append('d') #will add 'd' to the list
mylist.pop(2) # will remove items by position (index), remove the 3rd item
mylist.remove(x) # Remove the first item from the list whose value is x.
mylist.index(x) #return the index of the first item whose value is x
mylist.count(x) # Return the number of times x appears in the list.
list.insert(i, x) #will insert an item before element with index i.
```

We can view all the methods defined for a object by using the built-in function `dir`. For example:

```
L = [] # define a list object
dir(L) # view all the methods of the list object
```

6.2 List Comprehensions

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses. For example,

```
>>> ls=[1,2,-3,-4]
>>> [math.sqrt(x) for x in ls if x>0]
[1.0, 1.414]
```

The following code combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

6.3 String and string methods

Single or double quotation marks are used to define a string value:

```
In [17]: a = 'hello, world' # string
In [18]: a = "hello, world" # string
```

Each character in a string corresponds to an index and can be accessed using index notation (similar to a list):

```
In [19]: a[0]
```

```
Out[19]: 'h'
In [21]: a[0:6]
Out[21]: 'hello,'
```

String methods:

```
a = "hello, world" # string
a.split(",")  #Splits the string at the specified separator, and returns a list
a.find("o") #Searches the string for "o" and returns the position of where it was
found
```

Again, we can view all the methods defined for `string` object by using the built-in function `dir`. Or search "python string methods" online to find useful methods of string objects. Note that all string methods returns new values. They do not change the original string.

6.4 Tuple

Another data structure similar to list is tuple, which is an ordered collection of items enclosed in round-brackets (parentheses):

```
t=(1,2,"apple")
```

The round-brackets are optional.

Slicing and addressing a tuple are similar to those of a list. Like a list, we can loop through the tuple items by using a `for` loop. Unlike a list, Tuples are **unchangeable**. Once a tuple is created, we cannot change its values.

6.4.1 Tuple methods

```
mytuble.count("apple") # Returns the number of times a specified value occurs in a
tuple
mytuble.index("apple") # Searches the tuple for a specified value and returns the
index
```

6.5 Set

```
a={"dd", 1, (3,4)} #an items in set can be a tuple, but can not be a list
```

6.5.1 Access items in a set

We can not access an item in a set by referring to an index, since sets are unordered and have no index:

```
In [7]: a={"dd", 1}
In [8]: a[1]
TypeError: 'set' object does not support indexing
```

But we can loop through a set using `for` loop, or ask if a specified value is present in a set by using the `in` keyword.

```
myset = {"apple", ‘‘banana", ‘‘cherry"}
print("banana" in myset) #True
for x in myset:
    print(x)
```

6.5.2 Set methods

```
myset.add("apple") # adds an element to the set
myset.remove("apple") # removes a particular element from the set
myset.pop() # removes an random element from the set, retuns the removed item
```

6.5.3 Set comprehension

```
>>>s={v for v in 'abcdabcd' if v not in 'cb'}
>>> print(s)
{'a', 'd'}
>>> type(s)
<class 'set'>
```

In `Racket`, the above set comprehension is written as

```
(for/set ([v "ABCDABCD"] #:unless (member v (string->list "CB"))) v )
```

6.6 Dictionary

A dictionary is a collection of a pair of items enclosed in curly brackets:

```
d={"a":1, "b":2} #each element has two filds, key and value, separated by :
```

In other languages, data types similar to Python dictionaries may be called “hashmaps” or “associative arrays”.

Dictionaries can be built up and added to in a straightforward manner:

```
In [8]: d = {}
In [9]: d["last name"] = ‘‘Alberts"
In [10]: d["first name"] = ‘‘Marie"
In [11]: d["birthday"] = ‘‘January 27"
In [12]: d
Out[12]: {'birthday': 'January 27', 'first name': 'Marie','last name': 'Alberts'}
```

The type of keys of a dictionary can be string, int, float, and even a tuple. For example:

```
In [15]: A={(1,2):4, "b":5}
In [16]: A[(1,2)]
Out[16]: 4
```

It is interesting to note that referencing a dictionary item is very similar to referencing a list element if the keys are of int type.

6.6.1 Dictionary methods

```
d={"a":1, "b":2}
d.keys() # return all the keys of a dictionary
d.values() # return all the values of a dictionary
```

7 File Handling

The key function for working with files in Python is the `open()` function, which takes two parameters; *filename*, and *mode*, and returns a file object. For example:

```
f = open('t.txt', 'r')
```

There are four different modes for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist `"a"` - Append - Opens a file for appending, creates the file if it does not exist

`"w"` - Write - Opens a file for writing, creates the file if it does not exist

`"x"` - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

`"t"` - Text - Default value. Text mode

`"b"` - Binary - Binary mode (e.g. images)

7.1 Methods of file objects

A python file object has several predefined methods, such as `read`, `readline`, `readlines`, etc. For example:

```
f = open('t.txt', 'r')
txt = f.read() #readin the entire file, return a string
f.close() #close the file
f = open('t.txt', 'r')
txt1 = f.readline() #readin one line from the file, return a string
txt2 = f.readline() #readin another line from the file
f.close()
f = open('t.txt', 'r')
txt = f.readlines() #read the entire file, return a list of string (one
string=>one line)
```

A file object can also be converted to a list:

```
f = open('t.txt', 'r')
a = list(f) # return a list, the same as a=f.readlines()
```

A python file object is also an iterator, which means that we can loop over the file object:

```
f=open('t.txt', 'r')
for line in f:
    print(line, end='')
```

8 Python class and object

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. Define a class named `MyClass`, with a property named `x`:

```
class MyClass:
      x = 5
```

Now we can use the class `MyClass` defined above to create an object:

```
p1 = MyClass()
print(p1.x)  #use the dot notation to access a property:
```

Python Iterators

An iterator is an object that contains a countable number of values and can be iterated upon, meaning that we can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`. The `for` loop actually creates an iterator object and executes the `next()` method for each loop.

When developing this document, I read the following materials:

https://docs.python.org/

https://www.w3schools.com/python/

https://physics.nyu.edu/pine/pymanual/html/

9 Numpy and matplotlib

9.1 Numpy array

The data of a NumPy array are stored in contiguous block of system memory. This is the main difference between an array and a pure Python structure, such as a list, where the items are scattered across the system memory. This aspect is the critical feature that makes NumPy arrays efficient.

Practically all software has some bugs; it's a matter of frequency and severity rather than absolute perfection.

When a bug does occur, you want to spend the minimum amount of time getting from the observed symptom to the root cause.

Parallelism consists of performing multiple operations at the same time. Multiprocessing is a means to effect parallelism, Multiprocessing is well-suited for CPU-bound tasks: tightly bound for loops and mathematical computations usually fall into this category.

Threading is a concurrent execution model whereby multiple threads take turns executing tasks. One process can contain multiple threads.

While a CPU-bound task is characterized by the computer's cores continually working hard from start to finish, an IO-bound job is dominated by a lot of waiting on input/output to complete.

non-blocking function

Over the last few years, a separate design has been more comprehensively built into CPython: asynchronous IO, enabled through the standard library's asyncio package and the new async and await language keywords. (async IO is not a newly invented concept, and it has existed or is being built into other languages and runtime environments, such as Go, C#, or Scala.)

Async IO is not threading, nor is it multiprocessing. It is not built on top of either of these.

In fact, async IO is a single-threaded, single-process design: it uses cooperative multitasking.