Programming in Python 3

BY YOUJUN HU

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

1 Introduction

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together.

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

Python is case sensitive.

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

Python is lexical scoped, like most contemporary programming languages.

Python functions are first-class citizens, which means: they can be treated like any other variable, can be passed to a function, and can be returned from any other function.

Python, like most languages, is multi-paradigm: object-oriented, procedural (imperative), and functional.

Python's sytax emphasizes readability and therefore reduces the cost of program maintenance.

Python uses indentation for statement grouping. Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements (i.e. code blocks such as loops, branchings, function/class body). Indentation-based syntax is believed to enhances source code readability. Most programming languages use brackets for defining blocks (e.g., `C` and `Java` use curly-brackets, Lisp/Scheme uses round-brackets).

A python statement can be delimited by a newline or semicolon. Therefore semicolon can be used to put multiple statements in the same line. This works only for some simples statements and does not work for compound statements (e.g., loop) due to the requirement of the layout syntax. Semicolons at the end of a Python command is not necessary, but if used, can suppress some output in interactive modes.

Python tries to keep the number of keywords small and many things in practical programming are done by function calls, rather than keyword statements. For example, early terminating a script is by function calls, such as `exit()`, `sys.exit()`, and `quit()`.

The reserved words (keywords) in Python are as follows.

Value keywords: `True, False, None`

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

Control flow keywords: `if, elif, else, for, while, break, continue, pass, return, yield`

Structure keywords: `def, class, lambda, with, as`

Variable handling keywords: `del, global, nonlocal`

Import module keywords: `import, from, as`

Exception-handling keywords: `try, except, raise, finally, else, with, assert`

Asynchronous programming keywords: `async, await`

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

Some identifiers are only reserved under specific contexts. These are known as soft keywords. The identifiers `match`, `case` and `_` can syntactically act as keywords in contexts related to the pattern matching statement, but this distinction is done at the parser level, not when tokenizing.

A lexical analyzer breaks a file into tokens (a token is a string with an assigned and thus identified meaning). The stream of tokens generated by the lexical analyzer is then used as input to the parser.

Comments in python: (1) anything after a hash (`#`) in a line is a comment. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. (2) triple quotes introduce string literals, which can have multiple-lines, and thus can serve as multiple-line comments.

2 Run Python program

1 Interactive mode

In a Linux terminal, invoking `python` without any comand-line option will enter the interactive mode, where we can type python source code:

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

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 (in Linux) 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
```

3 Use python module/library

In Python, 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. Use `import` to get access to python modules. For example:

```
import foo
```

Then python will look for a file with name being `foo.py` first in the current directory and then in other directories assumed by python. If the file is found, it will be loaded. Here `foo` is called module name. More examples:

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

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. A user of a module can access a module's global variables using `modname.itemname`, no matter the item is a function or regular variable.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
from numpy import sin
```

which import a specific function to the current namespace. Another form of `import` is

```
from math import *
```

This imports all names (except those beginning with an underscore) defined in the `math` package directly in the calling module's namespace. In most cases Python programmers do not use this form since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

3.1 Install a new module

Command tool `python3-pip` can be used to install a new package. For example, in a linux terminal,

```
pip3 install numpy
```

will install the `numpy` package.

4 Objects, types, values, and name binding

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. Every object has an identity, a type, and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The `id()` function returns an integer representing its identity. The '`is`' operator compares the identity of two objects;

An object's type determines the operations that the object supports and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). Like its identity, an object's type is also unchangeable.

Some objects contain other objects; these are called containers. Examples of containers are tuples, lists, sets, and dictionaries.

The value of some objects can change. Objects whose value can change are said to be mutable; objects whose value is unchangeable once they are created are called immutable. The value of an immutable container object that contains a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the `id()` of its element are not changed. So, immutability is not strictly the same as having an unchangeable value.

An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

For immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists.

All python's objects were based on a C data structure and its variations no matter the object is a simple object such as an integer, i.e., primitive, or something more complicated such as a class.

On high level of abstraction, concepts in python can be categorized into two concepts: names and objects. Names refer to objects. Names are usually called variables. A name is introduced by name binding operation, which binds the name to an object, i.e, naming the object. The following constructs bind names: assignments, class definitions, function definitions, formal parameters to functions, import statements, for loop header, a capture pattern in structural pattern matching.

(The physical representation of a name is most likely a pointer, but that's simply an implementation detail. Name is actually an abstract notion at heart.)

Objects have individuality, and multiple names can be bound to the same object. This is known as aliasing in some languages. Let us first discuss the most basic name binding operation—assignements. The Python assignment operator is `=`. Python assignment statements do not return values. For example

```
a = 1
b = 1
```

Then `a` and `b` may or may not refer to the same object (this can be checked with the build-in function `id()`.)

```
a = []
b = []
```

Here two objects (empty lists) are created in memory, and are named as `a` and `b`, respectively. Therefore `a` and `b` refer to the different objects.

```
a = []
b = a
```

Here the second line give an alias to the object named `a`, rather than copying the objects. Therefore `a` and `b` refer to the same object.

Python's built-in types include bool, numerics, sequences, mappings, classes, instances, and exceptions. Numeric types includes int, float, and complex. Sequence types include list, tuple, string, and range. Mapping Types — dict,

sets,

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

4.1 Argument unpacking

Argument unpacking is the idea that a tuple can be split into several variables depending on the length of the sequence. For instance, you can unpack a tuple of two elements into two variables:

```
t = (2, 3)
a, b = t
```

In a for loop, argument unpacking can be used along the built-in zip(), which allows you to iterate through two or more sequences at the same time. On each iteration, zip() returns a tuple that collects the elements from all the sequences that were passed:

```
>>> first = ["a", "b", "c"]
>>> second = ["d", "e", "f"]
>>> third = ["g", "h", "i"]
>>> for one, two, three in zip(first, second, third):
...     print(one, two, three)
...
```

```
a d g
b e h
c f i
```

More examples of argument unpacking (structured assignments):

```
a,b,c = 1,2,3
(a, (b,c)) = (10, (20,30))
[x,y] = [2,3]
a,b,c = 'Hey'
p, *q = ''Hello'' # q will refer to a list containing characters after ''H''
```

This kind of structure assignments appear often in everyday coding, in a disguised way when several variables are assigned with values of a function which returns multiply items. For example:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(nrows=2, ncols=2)
```

Python adopts dynamical and strong typing system, where "dynmical typing" means type checking happens at run time. In dynamically typed languages, typing is associated with the object that a variable refers to rather than the variable itself (a variable is a pointer pointing to the location where a value is stored). Therefor there is no type declaration for variables in dynamically typed languages. And a variable pointing to an object of a type can be later used to point to an object of another type.

5 Flow control

5.1 Logical expressions

Relation operators and logical operators in python are:

Equals	`==`
Not Equals	`!=`
Less than	`<`
Less than or equal to	`<=`
Greater than	`>`
Greater than or equal to	`>=`
Belongs to	`in`
object identity	`is`

Table 1. Relation operators

logical and	`and`
logical or	`or`
logicl not	`not`

Table 2. Logical operators

The returned values by the above operators are of logical type (also called bool). (Logical type is a subclass of int.) Logical expressions can be used in several ways, most commonly in conditionals and 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")
```

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)
```

Another usage of the one-line `if` structure is in the list comprehension:

```
>>> original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
>>> prices = [i if i > 0 else 0 for i in original_prices]
>>> prices
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

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 those in Fortran and C.

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}
```

which is essentially a `while` loop discussed above.

Python for-loop is collection based loop, which means it is always used in combination with an iterable object, like a list or a range. With python `for` loop, we can execute a set of statements, once for each item in a sequence. For example:

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

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

`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.

The advantage of collection-based iteration is that it helps avoid the off-by-one error that is common in other programming languages.

Nested loops:

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print(x)
```

The `with` statement clarifies code that previously would use try...finally blocks to ensure that clean-up code is executed. The with statement is a control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods).

The classic example is opening a file, manipulating the file, then closing it:

```
with open('output.txt', 'w') as f:
    f.write('Hi there!')
```

The above with statement will automatically close the file after the nested block of code. The advantage of using a with statement is that it gaurantee that the file will be closed no matter how the nested block exits. If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler. If the nested block were to contain a return statement, or a continue or break statement, the with statement would automatically close the file in those cases, too.

6 Function

6.1 Define and call a function

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

```
def func_name ( arg1, arg2, argN ):
    some_codes
    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
```

Functions can return multiple items:

```
def func():
    return 'a', 3, (1,2,3)  # returns a tuple of 3 elements (str, int, tuple)
x1, x2, x3 = func()  # unpacks the tuple of 3 elements into 3 vars
# x1: 'a'
# x2: 3
# x3: (1,2,3)
```

6.2 Arguments are passed by assignment

Some languages (e.g. Fortran) handle function arguments as references to existing variables, which is known as pass by reference. Other languages (e.g. C) handle them as independent values, an approach known as pass by value.

Python assigns a unique identifier to each object and this identifier can be found by using Python's built-in `id()` function. It is ready to verify that actual and formal arguments in a function call have the same id value, which indicates that the dummy argument and actual argument refer to the same object.

Note that the actual argument and the corresponding dummy argument are two names referring to the same object. If you re-bind a dummy argument to a new value/object in the function scope, this does not effect the fact that the actual argument still points to the original object because actual argument and dummy argument are two names.

The above two facts can be summarized as "arguments are passed by assignment", i.e.,

```
dummy_argument = actual_argument
```

If you re-bind dummy_argument to a new object in the function body, the `actual_argument` still refers to the original object. If you use `dummy_argument[0] = some_thing`, then this will also modify `actual_argument[0]`. Therefore the effect of "pass by reference" can be achieved by modifying the components/attributes of the object reference passed in. Of course, this requires that the object passed is a mutable object.

To make comparison with other languages, you can say Python passes arguments by value in the same way as C does, where when you pass "by reference" you are actually passing by value the reference (i.e., the pointer)

In practical programming, returning multiple values from functions is usually better than employing the effect of passing by reference.

6.3 Arguments packing and unpacking

A formal argument (dummy argument), which appear in a function defintion, is often referred to as "parameter". The actual argument, which appears in a function call, is often referred to as "argument". When seeing the word "argument", we need to judge from the contex whether it refers to a formal argument or actual argument.

Besides standard positional arguments, there is a special formal argument that can accept a group of positional arguments and pack them into a single iterable object:

```
def my_sum(*args):
    result = 0
    for x in args:
        result += x
    return result

print(my_sum(1, 2, 3))
```

Here my_sum() takes all the parameters that are provided in the input and packs them all into a single iterable object (named `args` in this case). The name `args` does not mather. All that matters here is that you use the unpacking operator (*) before `args`.

**kwargs works just like *args, but instead of accepting positional arguments it accepts keyword (or named) arguments. (Like args, kwargs is just a name that can be changed to whatever you want. Again, what is important here is the use of the unpacking operator **.) Take the following example:

```
# concatenate.py
def concatenate(**kwargs):
    result = ""
    # Iterating over the Python kwargs dictionary
    for arg in kwargs.values():
        result += arg
    return result

print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Note that in the example above the iterable object is a standard dict. If you iterate over the dictionary and want to return its values, like in the example shown, then you must use kwargs.values().

To recap, in a function defintion, use *args and **kwargs to accept a changeable number of positional arguments and keyword argument, respectively.

What if you want to create a function that takes a changeable number of both positional and named arguments (keyword arguments)?

In this case, you have to bear in mind that order counts. Just as non-default arguments have to precede default arguments, so *args must come before **kwargs.

To recap, the correct order for your parameters is:

1. Standard arguments
2. *args arguments
3. **kwargs arguments

You are now able to use *args and **kwargs to define Python functions that take a varying number of input arguments. Let's go a little deeper to understand something more about the unpacking operators.

The single asterisk operator * can be used on any iterable that Python provides, while the double asterisk operator ** can only be used on dictionaries.

starred expression

The most straightforward way to pass arguments to a Python function is with positional arguments (also called required arguments).

the parameters given in the function definition are referred to as formal parameters, and the arguments in the function call are referred to as actual parameters.

Positional arguments must agree in order and number with the parameters declared in the function definition.

Keyword arguments must agree with declared parameters in number, but they may be specified in arbitrary order.

Default parameters allow some arguments to be omitted when the function is called.

Things can get weird if you specify a default parameter value that is a mutable object.

In Python, default parameter values are defined only once when the function is defined (that is, when the def statement is executed). The default value isn't re-defined each time the function is called.

When a parameter name in a Python function definition is preceded by an asterisk (*), it indicates argument tuple packing. Any corresponding arguments in the function call are packed into a tuple that the function can refer to by the given parameter name. Any name can be used, but `args` is so commonly chosen that it's practically a standard.

An analogous operation is available on the other side of the equation in a Python function call. When an argument in a function call is preceded by an asterisk (*), it indicates that the argument is a tuple that should be unpacked and passed to the function as separate values. Although this type of unpacking is called tuple unpacking, it doesn't only work with tuples. The asterisk (*) operator can be applied to any iterable in a Python function call.

Argument Dictionary Packing
Python has a similar operator, the double asterisk (**), which can be used with Python function parameters and arguments to specify dictionary packing and unpacking. Preceding a parameter in a Python function definition by a double asterisk (**) indicates that the corresponding arguments, which are expected to be `key=value` pairs, should be packed into a dictionary.

Argument dictionary unpacking is analogous to argument tuple unpacking. When the double asterisk (**) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments. Think of *args as a variable-length positional argument list, and **kwargs as a variable-length keyword argument list.

The bare variable argument parameter * indicates that there aren't any more positional parameters. This behavior generates appropriate error messages if extra ones are specified. It allows keyword-only parameters to follow.
To designate some parameters as positional-only, you specify a bare slash (/) in the parameter list of a function definition. Any parameters to the left of the slash (/) must be specified positionally.
The positional-only and keyword-only designators may both be used in the same function definition

```
>>> # This is Python 3.8
>>> def f(x, y, /, z, w, *, a, b):
...     print(x, y, z, w, a, b)
...

>>> f(1, 2, z=3, w=4, a=5, b=6)
1 2 3 4 5 6

>>> f(1, 2, 3, w=4, a=5, b=6)
1 2 3 4 5 6
```

In this example:

x and y are positional-only.
a and b are keyword-only.
z and w may be specified positionally or by keyword.

Use the `yield` keyword to send values back to the caller. When you call a function that contains a yield statement anywhere, you get a generator object, but no code runs. Then each time you extract an object from the generator, Python executes code in the function until it comes to a yield statement, then pauses and delivers the object. When you extract another object, Python resumes just after the yield and continues until it reaches another yield (often the same one, but one iteration later). This continues until the function runs off the end, at which point the generator is deemed exhausted

6.4 Namespace and Scope

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries.

The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name.

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function.

Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that you do not need to qualified the name with namespace name being suffix, i.e., you can directly use the name itself to access its refereence.

A variable defined inside a function’s body is known as a local variable. Formal arguments identifiers also behave as local variables. A variable created outside of functions is known as a global variable, which can be called a free variable from the perspective of a function.

Python adopts the lexical scope, which means the binding of a free variable inside a function can be infered without considering where the function is called (but the value of the free variable can depend on where the function is called because the value of the free variable can be modified somewhere).

Compared with statically typed languages, the scope of a variable in a dynamically typed language like python becomes a little subtle. 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` or `nonlocal`.

Therefore, to update a global variable inside a function, we need 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. 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.

In a nested scope, we can use keyword `nonlocal` to declare that a variable is from enclosing scopes (scopes enclosing the present scope) but not a global. In other words, `nonlocal` means “not a global or a local variable”.

The order in which Python looks up names is as follows. If you reference a given name, then Python will look that name up sequentially in the Local, Enclosing, Global, and Built-in namespaces. If the name exits, then you’ll get the first occurrence of it. Otherwise, you’ll get an error. This rull is often called LEGB rule.

What is the relationship between scope and namespaces in Python?

The terms, namespace and scope, can be used almost interchangeably because they overlap a lot in what they imply. A namespace is a dictionary, mapping names (as strings) to values. When you make a reference, like `print(a)`, Python looks through a list of namespaces to try and find one with the name as a key.

A scope defines which namespaces will be looked in and in what order. The scope of any reference always starts in the local namespace, and moves outwards until it reaches the module’s global namespace, before moving on to the builtins (the namespace that references Python’s predefined functions and constants, like `range` and `getattr`), which is the end of the line.

When we say x is in a function’s namespace, we mean it is defined there, locally within the function. When we say x is in the function’s scope, we mean x is either in the function’s namespace or in any of the outer namespaces that the function’s namespace is nested inside.

Whenever you define a function (using `def` or `lambda`), you create a new namespace and a new scope. The namespace is the new, local hash of names. The scope is the implied chain of namespaces that starts at the new namespace, then works its way through any outer namespaces (outer scopes), up to the global namespace (the global scope), and on to the builtins.

A scope refers to a region of a program from where a namespace can be accessed without a prefix.

Classes in Python do not introduce a new namespace, which is why class attributes must be qualified with the class name and why instance attributes must be qualified with the instance name.

Lexical scoping:

```
x = 1
def myfun():
    return x
x = 10
myfun()  # return 10
```

Q: Is the above behavior consistent with lexical scoping?

Q.Yes, it is. myfun is using the x variable from the environment where it was defined, but that x variable now holds a new value. Lexical scoping means functions resovle variables from the scope where they were defined, but it doesn't mean they have to take a snapshot of the values of those variables at the time of function definition. The code line `x = 1` can be dropped and the codes are still valid, i.e., `x` can be un-bound when the function is defined.

Lexical scoping means functions resolve free variables from the scope where they were defined, not from the scope where they are called.

The automatic destruction of unreferenced objects is called garbage collection.

The nonlocal statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. SyntaxError is raised at compile time if the given name does not exist in any enclosing function scope.

A global statement cause corresponding name to refer to the name in the global scope (create the binding in the global scope if there is a name binding for that name in the local scope).

6.5 Nested function, return function as value, closure

In Python, we can define a function inside the body of a function definition (function inside function, nested function). The inner functions have access to all outer variables in the enclosing functions. The inner functions can be called in the body of the enclosing function body. This feature is commonplace and can be found in most languages. What is interesting is that the inner function can be returned as the return value of the enclosing function. This feature is only available in languages that treat function as a first citizen. For example

```
def power_factory(exp):
    def power(base):
        return base ** exp
    return power

square = power_factory(2)
square(10)  #give 100
cube = power_factory(3)
cube(10) #give 1000
```

Variables like `exp` in the inner fucntion are called free variables. They are variables that are used in a code block but not defined there. When you return the inner function as the return value of the enclosing function, python needs to remember the values of these free variables (otherwise the returned function is meaningless). In other words, when you handle a nested function as value, the inner 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 function that carries information about its enclosing scope, even though its ecnclosing scope has completed its execution.

Another famous example of making use of closure is to generate the derivative function of a given function:

```
>>> def derivative(f, dx):
        def prime(x):
            return (f(x+dx)-f(x))/dx
        return prime
>>> dx=0.01
>>> mycos=derivative(math.sin,dx)
>>> mycos(2.0)
```

6.6 Function call vs keyword structure

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")
```

Similar to `exec`, `eval` is a build-in function, which evaluates an expression given as a string:

```
eval("1+2")
```

The difference between `exec` and `eval` is that `eval` returns a value while `exec` does not. `eval` can not be used for statements (such as assignment) while `exec` can be used for both statements and expressions.

Statements are different from expressions in that statements do not return results and are executed solely for their side effects, while expressions always return a result and often do not have side effects at all.

6.7 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)
```

Usually 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 [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
```

7 Class

Class concept is a way of bundling data and functionality together, and supporting extention (i.e., inheritance). Define a new class define a new type of object, allowing new instances of that type to be created. A class is a blueprint for the instances of that type. The process of creating the object of that type is called instantiation.

Compared with other programming languages, Python introduces class concepts with a minimum of new syntax and semantics. The following is a typical defintion of a class:

```
class ComplexNumber:
    scale = 1
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i
    def show(self):
        print(f'{self.real}+{self.imag}j')
        print(f'scale={self.scale}')
```

The above define a class named `ComplexNumber`, with two methods and three attributes named `scale`, `real`, and `imag`.

In practice, the statements inside a class definition will usually be function definitions. The function definitions inside a class must follow a peculiar form of argument list: The first dummy argument of a method is assumed by Python to be the object in question. The dummy object name is usually called `self`. This name is just a user convention: you can use an arbitry name here (i.e., what matters is its position in the argument list, not its name). We will follow this convention. (When the method is called, users must omit the object name from the argument list, and the name is figured out and inserted to the argument list by python behind the scene.)

In a method, attributes of a class instance are created or referred to by using the dot notation: `self.attribute`. In the above, `self.real` and `self.imag` are attributes of an instance, whereas `scale` is an attribute of the class. Note that, if the usual scoping rule applies (i.e., the class defintion forms a parent scope for its methods), the attribute `scale` should be assessible in the method by using the name `scale`. It turns out we can refer to `scale` but must use a different name: the attribute must be qualified with the class name, i.e., `ComplexNumber.scale`.

This indicates that the scope of names defined in a class block is limited to the class block. It does not extend to the code blocks of methods. This is one of the new rules introduced to facilitate class defintion, i.e., class defintion body is not used as a parent scope for methods. Otherwise, it would make class inheritance confusing: e.g., a method inherited by a subclass would have access to the subclass scope.

Note that, in the above, if we

The idea is that self.x first looks into the instance for the attribute x, and when that fails, it looks into the class itself.

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

```
a = ComplexNumber(2,3)
```

Class instantiation uses function notation with the class name serving as a function name. The returned value is a new instance of the class.

If there is a method named `__init__`, which is ture for the example shown above, it will be used by python as a constructor, which means this method is automatically called when we create an instance of the class.

When the method is called, users must omit the object name from the argument list, and the name is figured out and inserted to the argument list by python behind the scene.

Class functions that begin with double underscore __ are called special functions as they have special meaning. Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Next, examine the instance we created above:

```
a.show() #the object a is implitcitly provided as the first argument
print(a.real)
```

In Python, attributes may be referenced/created by methods as well as by users of an object. There are no "private" instance variables (variables that can only be accessed within methods of an object). In other words, it is imposible to enforce data hiding in python — it is all based upon convention. The convention (followed by most Python code) is: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a method or a data member).

Note that clients may add data attributes of their own to an instance object. As long as name conflicts are avoided, adding new data abttributs does not affect the validity of the methods.

The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: functions and modules imported into the global scope can be used by methods.

class attribute vs. instance attribute

Attributes of an object can be created on the fly.

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method or static method.

Use the @classmethod decorator or the classmethod() function to define the class method

Use the @staticmethod decorator or the staticmethod() function to define a static method.

we'll find some good reasons why a method would want to reference its own class.

A language feature would not be worthy of the name "class" if it does not support inheritance.

Finally, notice that the class .__dict__ and the instance .__dict__ are totally different and independent dictionaries. That's why class attributes are available immediately after you run or import the module in which the class was defined. In contrast, instance attributes come to life only after an object or instance is created.

The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods.

In general, when you're writing object-oriented code in Python and you try to access an attribute, your program takes the following steps:

Check the instance local scope or namespace first.

If the attribute is not found there, then check the class local scope or namespace.

If the name doesn't exist in the class namespace either, then you'll get an AttributeError.

Although classes define a class local scope or namespace, they don't create an enclosing scope for methods.

Classes themselves are objects (class objects), which means that a class name can be rebound to new names. This makes importing easy, where we can rename a class name to a new simple name.

Most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

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.

8 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
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 optional **round-brackets**, 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.

Lists, tuples, and strings are subscriptable, but sets are not. Attempting to access an element of an object that isn't subscriptable will raise a TypeError.

An object is mutable if its structure can be changed in place rather than requiring reassignment:

Lists and sets are mutable, as are dictionaries and other mapping types. Strings and tuples are not mutable. Attempting to modify an element of an immutable object will raise a TypeError.

8.1 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. I will not be using Python arrays at all. Therefore, whenever I refer to an "array," I 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])
```

8.1.1 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.

8.1.2 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
```

8.2 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)]
```

List comprehensions are also more declarative than loops, which means they're easier to read and understand. Loops require you to focus on how the list is created. You have to manually create an empty list, loop over the elements, and add each of them to the end of the list. With a list comprehension in Python, you can instead focus on what you want to go in the list and trust that Python will take care of how the list construction takes place.

8.3 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 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.

8.4 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.

8.4.1 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
```

8.5 Set

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

A set itself may be modified, but the elements contained in the set must be of an immutable type. Therefore a list can not be an element of a set.

Set items are unordered, unindexed, and do not allow duplicate values. Set items are unchangeable, but you can remove items and add new items.

8.5.1 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)
```

8.5.2 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
```

8.5.3 Set comprehension

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

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

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

8.6 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.

8.6.1 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
```

The following is a summary of useage of various brackets in python:

- square brackets `[]` are used in defining lists, list comprehensions, and for retriving elements from lists/tuple/dicts, where retriving can be indexing/slicing/lookup. In numpy, `[]` are used for arrays.
- round brackets `()` are basically used to group things. Specifically, `()` are used in function definitions/calls to gather arguments, generator expressions, defining order of operations, and tuples, where `()` can be omitted if there is no ambiguity.
- curly brackets `{}` are used in defining the two hash table types– dictionaries and sets.

9 File Handling

9.1 Create file object

Python builtin function `open()` 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)

9.2 Methods of file object

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

```
f = open('t.txt', 'r')
txt = f.read() #readin the entire file, return a string
f.close()
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='')
```

10 Numpy and matplotlib

NumPy is the reason why Python stands among the ranks of R, Matlab, and Julia, as one of the most popular languages for doing STEM-related computing. It is a third-party library (i.e. it is not part of Python's standard library) that facilitates numerical computing in Python by providing users with a versatile N-dimensional array object for storing data, and powerful mathematical functions for operating on those arrays of numbers. NumPy makes use of a process known as vectorization, that enables a degree of computational efficiency that is otherwise unachievable by the Python language.

The impact that NumPy has had on the landscape of numerical computing in Python is hard to overstate. Whether you are plotting data in matplotlib, analyzing tabular data via pandas and xarray, using OpenCV for image and video processing, doing astrophysics research with the help of astropy, or trying out machine learning with Scikit-Learn and MyGrad, you are using Python libraries that bare the indelible mark of NumPy. At their core, each of these libraries depend on NumPy's N-dimensional array and its efficient vectorization capabilities. NumPy also fundamentally impacts the designs of these libraries and the way that they interface with their users. Thus, one cannot leverage these tools effectively, and cannot do STEM work in Python in general, without having a solid foundation in NumPy.

10.1 Numpy array

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`. 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.

We can create a NumPy ndarray object by using various functions, such as `np.array()` and `np.zeros()`.

`ndarray.tolist()` Return a copy of the array data as a (nested) Python list.

```
x=np.array([1,2,3,4])
y=np.array([5,6,7])
```

```
XX, YY = np.meshgrid(x,y)
```

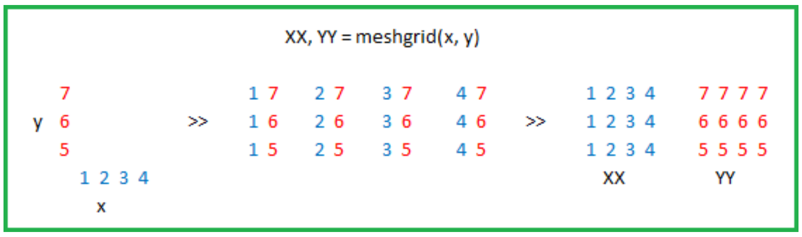

Figure 1.

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.

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.

All Python (intepretor) releases are open source. Python works on many platforms (e.g. Linux, Windows, Mac). Python has lots of libraries, documentation, and an active community.

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/