Technical Note. Some Python3 Details (Draft)

Konstantin Burlachenko, et al.

King Abdullah University of Science and Technology, Thuwal, Saudi Arabia.

Correspondence to: konstantin.burlachenko@kaust.edu.sa

Revision Update: July 06, 2023

Introduction

The web search engine Google returns around $6.95 \cdot 10^8$ answers from the search request "Python Tutorial". One of these tutorials is worthwhile to mention explicitly - it is a tutorial on the subject of this programming language originally written by *Guido Van Rossum* - https://docs.python.org/3/tutorial/index.html, which is located on the website of the most popular python distribution CPython. Another Python distributions Jython, Python for .NET are less popular.

Python is a popular scripting language which can be observed from:

- Tiobe-Index
- Google-Trends

On this note authors with some mix of backgrounds, but primarily CS background wants to achieve several goals:

- Share some technical details about some language features. It can be good material after reading Official Python Tutorial, if you do not have to get all important from the Official Python Tutorial.
- Share vision which system aware people CS/EE people think about the subject
- Provide a compact survey about several tools

We hope to make it will be an interesting reading for you. So let's go.

What is a Python

Python is an interpretable scripting language. Python has been designed originally only as a replacement for Bash, which has been described in that Blog Post written by original author:

> "...My original motivation for creating Python was the perceived need for a higher-level language in the Amoeba project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these in the Bourne shell wouldn't work for a variety of reasons. The most important one was that as a distributed micro-kernel system with a radically new design, Amoeba's primitive operations were very different (and finer-grain) than the traditional primitive operations available in the Bourne shell. So there was a need for a language that would "bridge the gap between C and the shell..." - Guido van Rossum.

This language does not have any Standards. The only available language standard is Language reference: https://docs.python.org/3/reference/.

The Python (and any interpreter) parses the program's text (source code) line by line (that is represented or in text form or extremely high-level instructions).

The standard implementation Python interpreter is CPython. It is called CPython because it has been implemented in C/C++. This software can be downloaded from https://www.python.org/.

Where to learn about Python If I have never heard about it?

There is exist a book written by Guido van Rossum (original author of that language). It's a pretty big and nice written tutorial: https://docs.python.org/3/tutorial/

- Python Language Reference: https://docs.python.org/3.8/reference/index.html
- Detailed description of different built-in functions for user-defined class: https://docs.python.org/3/reference/datamodel.html#emulating-callable-objects -
- Built-in types: https://docs.python.org/3/library/stdtypes.html
- Python Language Reference: https://docs.python.org/3.8/reference/index.html

Source of Confusion

Python is a useful and nice *scripting language*, which of course more comfortable to use compared to Perl or Windows Batch and sometimes more comfortable to use compare to Bash. In our experience Bash is still more easy to use if the script contains executing external processes for different tasks and less logic.

Python language has a very fast learning curve which opened the door fortunately for people not only in CS to create useful scripts in day-to-day life.

> Python is a programming language which...

Unfortunately, if you are around people with CS/EE/System/Compilers backgrounds it may be the case that these people will make a statement: **"Python is not a programming language"**.

To state that Python is *Programming Language* we need to define what is (1) programming and what is a (2) language.

One vague (and market) meaning is that Language expresses ideas and you don't care at all how these ideas are materialized.

But it's not the only definition. The message which one scientist B. Stroustroup (author of C++) tries to bring people at least for already 3 decades that *Programming Language* is the language that gives you a way to describe the algorithm and this algorithm will be executed in the computer (compute device). With this very strict definition Python, Java, C# what is executed is not your program. It's an interpreter in the case of Python and runtime coupled with a Just-In-Time(JIT) compiler for another two languages. If you only start with Programming or if you have no experience in Compilers/OS/Systems you may not see the difference, but their difference is fundamental even to catch this difference you should have some background.

It does not say that this language is incorrect, but I hope you should understand that it's not the computer that executes this program, but what is executing is another (one more) level of abstraction.

"Python is a general purpose programming language".

This is a very strong statement. And it can be three points of view on such a statement with anybody who has CS background:

1. By general-purpose programming language you mean that you can create any Algorithm. If this is a definition, then it's correct. Only DSL languages constructed for special purposes maybe have a lack of being Turing Complete.
2. If by General Purpose you mean that you can use it across a big amount of domains, then how big is big? This is a pretty vague definition.
3. If by General Purpose you mean you can create programs for different computing elements in the Computer, then it's almost *wrong*. You should have big flexibility with what you're doing. Second Python by design is the replacement for Bash, it's not a replacement for C or C++, or any other traditional language. You can not create drivers for your devices.

"Python is more elegant and short than C++".

This is true if look into lines of code, however the first thing in creating algorithms - they should be correct. In our experience, what is interesting after some amount of code there is a very strange asymmetry that you will observe once you will create projects in Python and C++ with 40K lines of code and more.

With such a big size (lower bound) Python is not even close to C++. The thing is that the absence of compilers and toolchains which you need to install and spend time using compile, just-in-time compile languages will hurt you. There are very well-developed tools.

Of course, there is a tool for Python that helps pylint, but a compiler and linker are far more powerful tools for detecting errors than a static analyzer.

"Python is everywhere".

This overstatement can also be read from Python Tutorial. Please be aware that (any) interpretable language which exists or will be created in the future fundamentally will have the following downsides:
C++ Technical Note / Downsides of Interpretable Languages
fundamentally.

Background: Programming Languages

How typical compute device is working

It's important to understand that computational devices do not execute code in Java, C#, Matlab, or Python.
Computers can not do it. It is what happening in reality.

The real computation devices execute binary code compiled in the form of Instruction Set Architecture(ISA). A simplified compute CPU device has computation cores that execute arithmetic operations via reading arguments and writing results into some form of memory. Memory allows to store input and output results.
Control units that control the execution of these low-level commands.

Finer details about how computers are working can be obtained from *System Architecture*, *Performace Engineering* courses and books.

Programming Languages Taxonomy

There are a lot of programming languages these days. One way to analyze programming languages is from the angle of the implemented type system:

- **Static Type System** - statically typed system languages are those in which type checking is done at compile time.
- **Dynamic Type System** - dynamically typed languages are those in which type checking is done at runtime (execution time).
- **Strong Type System** or **Strong Type safety** - implicit type casting is prohibited
- **Weak Type System** or **Weak Type safety** - implicit type casting is allowed.

C and C++ have a *Strong Static Type System*.

Examples of Languages with *Weak Type System* are Javascript and Perl.

Ruby and Python have a *Strong Dynamic Type System*. This means that types are inferred at runtime (dynamic type system), but implicit conversions between types are not allowed (strong type system).

What is Object Orientated Programming (OOP)

Please take a good course or read good books. Object-Orientated-Programming requires some special way to organize code, but it does not force to have a `class` keyword in Language.

Please check this Appendix if you're curious about what is OOP from a Computer Science point of view:
C++ Technical Note/Object Orientated Design
.

Phylosophy of Python

If you want to understand some hidden principles built-in Python language then read from the following command:

```
python -c "import this"
```

This provides a point of view of the world from the point of view of Python language.

How to start Interpreter

Based on [1] the Python interpreter operates somewhat like the Unix shell. In general, there are three ways to start the execution of a Python interpreter:

1. When it is called with standard input connected to a device that can emit symbols - it reads and executes commands interactively.

2. When a Python interpreter is called with a file name argument or with a file as standard input, it reads and executes a script from that file. For details see Python Tutorial / Interpreter
3. Next way of starting the interpreter is by calling

   ```
   python -c command [arg] ...
   ```

   It executes the statement(s) in the command, analogous to the shell s `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote the command in its entirety.
4. Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for the module as if you had spelled out its full name on the command line.

The script name and additional arguments are turned into a list of strings and assigned to the `sys.argv` variable.

- When no script and no arguments are given, `sys.argv[0]` is an empty string.
- When the script name is given as `-` it means standard input, and `sys.argv[0]` is set to '-'.
- When you execute a Python interpreter with the `-c` command used, sys.argv[0] is set to `-c`:

  ```
  python -c "import sys; print(sys.argv)"
  ```

- When the `-m` module is used, `sys.argv[0]` is set to the full name of the located module.

Language Constructions and sources of confusion for people with another languages background

Comment about events

Some programming languages like C# contain native support of event-based communication between objects to support Object Orientated Programming, however, Python (and C, C++) programming languages do not contain a native event-based system, even though there are frameworks on top of it that supported that (for example Qt).

Meaning of Object

Now there is a clash of terminology if you have C++, Java, or C# background. In Python, everything that takes up memory in some form is called the **object**. In other languages (Java, C#, C++) object is an instance of the class. So an object in Python terminology is:

- Class instances
- Exotic built-in objects (e.g. files)
- Fundamental built-in data types (e.g. integers)

It's not true that all things are classes in Python.

Python does provide access to variables by value

In Python, there are only references to objects. Python has named references to objects. It has no variables available to the programmer for reference by value. Functions return and accept arguments by reference in the terminology of C++, C#, and Java.

Moreover, there is a very strange thing, which was in Python 2.* and still is in the language about default argument. For the default argument - the default value itself is passed by reference and *calculated once*. This (unfortunately) creates an implicit global default value object. For C++ this is not observed, because arguments are passed by value, in the usual way of writing arguments, and is just a shorthand.

Context in Python is the same as scopes (in C++)

The context in Python is a concept similar to C++'s scope. However, in Python, you can not create scopes inside the function. For example if inside the function you have a nested loop, then its level of nested loop **does not** introduce a new scope, as a consequence the indexing variable will be rewritten.

In fact, in Python, there are only 4 contexts (scopes):

- **Local Context/Scope.** The context inside the current function. Importantly, a new loop with a new indentation does not introduce a new local scope.
- **Enclosing Context/Scope.** Locally declare a function inside a function. Variables from the parent function will be implicitly accessible to the children.
- **Global Context/Scope.** Variables from the top level of the module.
- **Built-in Context/Scope.** Built-in variables and functions built-in into the interpreter.

Pointers and Object Reference Equality

> Object in Python is everything that takes memory. Fundamentally in Python Object equality can test in one of two types:

> **Value equality** - testing which takes a reference to objects. And what is tested the objects have the same content. The programmer can define what this means via defining `__eq__` in your class. (This operator is the same as `==` in C++)

> **Identity equality** - testing when two references are referenced to the same object. The programmer can not determine what it means, it is defined by the language. Such type of equality is used during the use of operators `is`, `is not` or built-in function `id()`.

In Python 2/3 there are no pointers. However, there is a built-in function called `id`. In fact `id(obj)` according to [1],[2] `id` means the address of an object in the interpreter's memory. In principle that two objects refer to the same memory can be checked in the following way:

```
id(x) == id(y)
```

However in reality you will not meet this code too much. If you need to compare references to an object, then this action is typically performed by using the operators:

- `is`
- `is not`

They have the same semantical meaning as using `id()`.

> If you C++ background. The fact that id(x) is the memory address of object "x" is a CPython implementation detail. CPython can change this in the future. It's also hard to say what is standard and what is not - there is no standard for Language.

All class methods are virtual (in terms of C++)

Without loss of generality, we can assume that in Python all methods of all classes are `virtual`. But actually, there are no virtual methods and virtual keyword and virtual tables in Python at all mentioned in [1] or [2].

In fact, in Python methods are attributes. They are bound at runtime and executed dynamically. In very specific circumstances you can manually remove, get, and set attributes:

- delattr() - remove an attribute
- hasattr() - checks if an attribute exist.
- setattr() - set the value of an attribute
- getattr() - get the value of an attribute

There is also no division of fields into public/private/protected as it is in C++ and there are no different types of inheritances. What is absent in Python is a limited type of support for `private` names, which we will describe later.

Interfaces and Protocols

In Python languages, you will not find the keyword `interface` such as in Java/C# or pure virtual class methods as in C++.

In contrast, Python does not have interfaces as a language concept.

Instead of a specific (and robust) interface Python and other scripting languages uses what is known as Duck Typing.

This concept is described typically in the following way:

"If it walks like a duck and it quacks like a duck, then it must be a duck".

In Python, any object may be used in any context until it is used in a way that it does not support. The AttributeError will raise.

If your class is in Python and you want the objects of your class can be used in specific language construction (like) iteration you should support **protocol**. The protocol may not be a formal name, however, during the conversation you will hear about it:

- **Container protocol.** Built-in container types in Python
  tuple, list, str, dict, set, range, and bytes all support `in`, and `not in` operations for them. Support of this is named container protocol. For your types, you should define

  ```
  def __contains__ (self, item)
  ```

- **Sized protocol.** Built-in container types in Python support `len(x)` operator. For your types, you should define:

  ```
  def __len__(self)
  ```

- **Iterable protocol.** Built-in container types in Python support the iter(x), next(x) operators.

  Such objects can be used in for-loops:

  ```
  for I in iterable: smth(i)
  ```

  For your types you should define:

  ```
  def __iter__(self)
  def __next__(self)
  ```

  More details: https://docs.python.org/3/tutorial/classes.html#iterators

- **Sequence protocol.** Except dict, all Built-in container types in Python support indexing. It's known as sequence protocol. For your types, you should define **getitem**, **setitem**, **delitem**. Once you will define this operator you can call:

```
x [integral_index]
x.index(someValue)
x.count(someValue)
produceReverseSeq = reversed(x)
```

What is a False statement ?

The following expressions are considered as `false` in Python:

- None
- 0
- Empty sequence. I.e. sequence which `len() == 0`

Possible Python benefits which are absent in most Languages

It very depends on your point of view and your style, but there is a point of view where the following things are benefits. Especially if you have limited time for a project:

1. More correct code from a style point of view
2. Automatic cross-platform serialization "pickling" for the user and built-in types from the Python standard library.
3. A lot of free and commercial IDE with intellisense.
4. Python has a built-in debugger.
   The parsing of the function is performed only at the moment of the direct call. To run the script with a debugger call `python -m pdb scriptname.py`. After this, you will have the ability to insert text commands in the interactive shell with pdb commands.
5. Python use <`.`> token to separate packages/subpackage/class like C# and Java. Packages from Perl are called modules in Python.

About classes inside Python

Python has limited support for private attribute objects when naming attributes as `__attributename`. In this case, the interpreter performs name mangling. In the end, this attribute will be `_classname__attribute`.

In C++ terminology Python classes have the following characteristics:

1. Normal class members (including the data members) are public except for some small support for Private Variables.
2. All member functions/methods are virtual.
3. Like in C ++, most built-in operators with special syntax (arithmetic operators, etc.) can be redefined for class instances.
4. Python Data attributes correspond to "data members" in C ++.
5. Python supports multiple inheritance, and exceptions.
6. In Python (and in Perl) there is not exist such term as function overloading.

Key differences with C++

- Assignments do not copy data - the language just binds names to objects.

- In Python class data attributes override method attributes with the same name. To avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts.
- Nothing in Python makes it possible to enforce data hiding - it is all based upon convention.
- In Python, when using logical connectives, the result of a compound expression is equal to the result of the last subexpression evaluated. In C and C++ the result of a Boolean expression with short-circuiting is always an integer 0 or 1:

```cpp
#include <iostream>
int main() {
    std::cout << (12 || 2);
    return 0;
}
// Out: 1
```

```python
#!/usr/bin/env python3
print(12 or 2)
# Out: 12
```

- Python has the `elif` keyword. For Python, it's crucial because their syntax rules require creating indentation between `else` and `if`. C++ Language is more flexible, and such a keyword is absent in C/C++ but is presented in C Language Preprocessor. Example:

```python
if x < 0:
    pass
elif x == 0:
    pass
else:
    pass
```

- Python has a break statement in the else branch. If you did not read Python Tutorial [1] it can be the case that you never heard about it. In programming languages and fact in scripting languages (such as Python) such a concept is absent.
  The break statement, like in C/C++, breaks out of the innermost enclosing iteration loop. Loop statements may have an `else` path. This logic is executed when the loop terminates through exhaustion of the iterable or when the condition becomes false. See Also: https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops
- In Python there is the "Exotic" operator `**` used. This operator can raise integers, real, and complex numbers to specific power. Such a built-in operator is absent in C++.
- Unlike many languages, Python does not have unary post(postfix) increment (`x++`) or decrement (`x--`) operators.

Python Technical Basics

In the terminology of Programming Languages, **tokens** are separate words of a program text. One easy case is when such words (tokens) are split between each other by spaces. In most programming languages, the tokens fundamentally can be one of the following types:

- Operators
- Separators
- Identifiers
- Keywords
- Literal constants

Source file organization

The line `#!/usr/bin/env python3` in well-developed scripts is presented and it's called sha-bang. It has a long history for Unix/Linux OS-es. For Windows, it's possible to use it as well.

A binary named `py` is a launcher with performs a choice of a used interpreter if using sha-bang under Windows.

Source file encoding

The source file is represented in characters. Characters can be in any supported encoding https://docs.python.org/3.13/library/codecs.html#standard-encodings. By default, Python source files are treated to be encoded in UTF-8.

To declare an encoding other than the default one, a special comment line should be added as the first line of the file or a second line in the file if the first line is sha-bang.Example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Logical Lines and Parsing

Physical lines in source code are physical different lines in the text inside the file.

A Python program is read by a parser. The physical lines in the source text are not necessarily equivalent to the logical lines of the source program. A logical line of a program is constructed from one or more physical lines by following the explicit or implicit line-joining rules:

- When a physical line ends in a backslash `\` character that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character.
- A physical and logical line that contains only spaces, tabs, and possibly a comment is ignored by the parser.
- Expressions in parentheses, square brackets, and curly braces can be split over more than one physical line without using the backslash symbol `\`.

The parsing of Python source code (e.g. of the function) is performed only at the moment of the direct call of this function.

Comments

Comments in Python start with the hash character `#` and extend to the end of the physical line.

Operator Precedence

Information about operator precedence is available here:
https://docs.python.org/3.13/reference/expressions.html#operator-precedence

Simple built-in types

- **Ellipsis.** This type has a single value namely There is a single object with this value. This object is accessed through the literal `...`. If use this expression in condition then `...` is implicitly converted into `True`.
- **NoneType.** This type has a single value and there is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations. Also, it is returned from functions that do not explicitly return anything.
  If use this expression in condition then `...` is implicitly converted into `False`.

- **Integers (int)**. The type which is used to represent integers in Python does not have any fixed number of bits. Instead, it has a varying size. In this sense, an integer has an unlimited range. Practically you can hold as big an integer as you want until you start having problems with virtual memory.
- **numbers.Real (float).** These represent machine-level double-precision floating point numbers. From Documentation: *"...there is no reason to complicate the language with two kinds of floating point numbers..."* It's the design choice of a language. Of course, for people involved in numerics, such a statement is deeply wrong and sometimes laughable only. Documentaion: https://docs.python.org/3.13/reference/datamodel.html

Simple Statements

A simple statement is comprised of a single logical line. Several simple statements may occur on a single line separated by semicolons. Example:

```
a=1;b=2;
```

See Also: https://docs.python.org/2/reference/simple_stmts.html

Compound Statements

Generally, compound statements are written on multiple logical lines using indentation, although sometimes if a compound statement has only one branch in it, and the body of it contains only simple statements, it can be written on a single logical line. Example:

```
if 1: print (2); print (2); #legal python code
```

Empty(Pass) Statements

The `pass` statement does nothing similar to C++ `;` statement.

Division of numbers in Python

The division operator in Python `/` always returns a float. To do floor division and get an integer result you can use the `//` operator.

To calculate the remainder you can use `%` similar to C/C++/Java.

Comparision of Containers

Sequence objects (typically) can be compared to other objects with the same sequence type. The comparison uses lexicographical ordering.

Strings

Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...").

Printing

The standard way to print something in Python 3 is by utilizing the built-in print() function. There is a style for printing with a C# style string format:

```
print("Hello {1} / {0}".format(12,14))
```

The default behavior is to output the line into `stdout` with a new line. If you don't want to output a new line then in Python 3 you can specify the end character in the following way:

```
print("hello",end='!')
```

If you want to print not to `stdout`, but to `stderr` it can be attained in the following way:

```
import sys
print("hello {0}".format("world"), end='\n',file=sys.stderr)
```

You can use various formatting rules described here to configure output:
https://docs.python.org/3.11/library/string.html#formatspec

Since Python 3.6 is in the interpreter, a string interpolation feature has been added (https://www.python.org/dev/peps/pep-0498/). It is presented in Bash and similar scripting languages.
Example:

```
a = 123
print (f"Hello {a}")
```

The f-string forms what is known as a formatted string literal. To use formatted string literals, you should begin a string with `for` F` before the opening quotation mark.

Inside this string, you can write a Python expression between `{` and `}`. For example, you can refer to variables or literal values. During using f-string you can pass an integer after the `:`. It will cause that field to be a minimum number of characters wide. This is useful for making columns line up. Example:

```
a = 123
print (f"Hello {a:10}")
```

Next, there is one extra feature. The `=` specifier can be used to expand an expression to:

- Text of the expression in text form
- An equal sign
- Representation of the evaluated expression

Example:

```
a = 123
print(f"{a=}")
```

String interpolation (named formatted string literal) is evaluated during script execution. In terms of type, they are just `str` and it's not a new type. Implementing in C++ or any compiling language such functionality is impossible (of course if you don't want to write your interpreter inside your project).

Finally, old string formatting from Python 2 which is still supported in Python 3 for C style `printf()` C++ specification. Example:

```
print("%i %.2f" % (1, 0.12345))
```

Enumeration and Loops

Python's `for` statement iterates over the items of any sequence. If you do need to iterate over a sequence of numbers, the built-in function range() function will help you with that:

```
for i in range(5):
    print(i)
```

The object returned by `range()` behaves as if it is a list, but it is not. It is an object of class `range` which returns the successive items of the desired sequence when you iterate over it. It does not make the list, thus saving space.

When you are looping through dictionaries you are enumerating through `keys`. If the key and corresponding value are important for you, then can retrieve the value and key at the same time. For doing it you should use the items() method in the dictionary built-in type.

Next, when you are looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function. Example:

```
for  index, value in enumerate([10,11,23]):
  print(index, value)
```

To loop over a sequence in reverse, first, specify the sequence in a forward direction and then call the `reversed()` function. To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered. Using `set()` on a sequence eliminates duplicate elements because it creates the set.

More on Conditions

The conditions used in `while` and `if` statements can contain any operators.

- The comparison operators `in` and `not in` are membership tests.
- The operators `is` and `is not` compare whether two objects are the same object in memory.

What can confuse people with C++/Java/C#/C backgrounds is that comparisons can be chained. For example: `a < b == c` tests the following *"a is less than b* ***and*** *b equals c"*. Comparisons may be combined using the Boolean operators `and` and `or`, `not`. The Boolean operators `and` and `or` are so-called short-circuit operators and are analogous to `&&` and `||`. Python supports the same set of the bitwise operator as it in C/C++ languages (See operator precedence) and it inludes: `&`, `|`, `<<`, `>>`, `~`.

In Python when you use it as a general expression (not necessarily Boolean), the return value of a short-circuit operator is the last evaluated expression (which is not the case in C++ where you don't obtain this information).

Python Technical Details. Basics

Please if you mature enough with Python then it's better for you to skip this section or read it fast.

Introspection of varions Information

One way to get information about interpreter version:

```
python --version
```

Collect information about installed interpreter and system from the intepreter itself:

```
import os, platform, socket
print("================================================")
print("Information about your system")
print("================================================")
print(f"Python interpretator: {sys.executable}")
print(f"Python version: {sys.version}")
print(f"Platform name: {sys.platform}")
print("================================================")
print(f"Current working directory: {os.getcwd()}")
(system, node, release, version, machine, processor) = platform.uname()
print(f"OS name: {system}/{release}/{version}")
print(f"Host: {socket.gethostname()} / IP:
{socket.gethostbyname(socket.gethostname())}")
```

Basic data types

Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)       # Prints "4"
x *= 2
print(x)       # Prints "8"
y = 2.5
print(type(y))                 # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Bool variables and operators

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

Booleans are the results of comparisons like:

```
a = 3
b = 5
c = 7
print(a == a)     # Prints "True"
print(a != a)     # Prints "False"
print(a < b)      # Prints "True"
print(a <= a)     # Prints "True"
print(a <= b < c) # Prints "True"
```

Strings

Python has great support for strings:

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.

print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"

hw = hello + ' ' + world  # String concatenation
print(hw)                 # Prints "hello world"

hw12 = '%s %s %d' % (hello, world, 12)  # C/C++ sprintf style string formatting
print(hw12)                             # prints "hello world 12"

hw21 = '{} {} {}'.format(hello, world, 21)  # formatting with format function in C#
style
print(hw21)                                 # prints "hello world 21"
hw13 = '{} {} {:.2f}'.format(hello, world, 1 / 3)  # float formatting
print(hw13)                                        # prints "hello world 0.33"
hw3 = f'{hello} {world} {1 / 3:.2f}' # the f-strings
print(hw3)                           # prints "hello world 0.33"
```

String objects have a bunch of useful methods, for example:

```
s = "hello"
print(s.capitalize())           # Capitalize a string; prints "Hello"
print(s.upper())                # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))               # Right-justify a string; prints "  hello"
print(s.center(7))              # Center a string; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances; prints "he(ell)(ell)o"
print('  wo rld '.strip())      # Strip surrounding whitespace; prints "wo rld"
```

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples. Containers are devoted for storing values.

A list is the Python equivalent of an array (conceptually, even inside Python interpreter it's really implemented as a list), but is resizeable and can contain elements of different types:

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists. This is known as slicing. We will see slicing again in the context of numpy arrays.

```
nums = list(range(5)) # range is a function that creates a list of integers
print(nums)           # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])      # Slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])       # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])       # Slice from the start up to index 2; prints "[0, 1]"
print(nums[:])        # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])      # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]    # Assign a new sublist to a slice
print(nums)           # Prints "[0, 1, 8, 9, 4]"
```

Dictionaries

A dictionary stores (key, value) pairs. You can use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an existing element does nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

Loops

You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

If you want access to the index of each element and element itself within the body of a loop, use the built-in `enumerate` function:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print(f'#{idx+1}: {animal}')
```

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print(f'A {animal} has {legs} legs')
```

If you want access to keys and their corresponding values, it's better to use `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
```

Iterating over a set has the same syntax as iterating over a list. However since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
```

List Dictionary and Set comprehensions

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)
```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list. One of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)                              # Create a tuple
print(type(t))                          # Prints "<class 'tuple'>"
print(d[t])                             # Prints "5"
print(d[(1, 2)])                        # Prints "1"
```

Unpacking (Maybe Advanced)

All containers can be unpacked as follows:

```
t = (3, 2, 1)
a, b, c = t  # unpacks the tuple t; prints "3 2 1"
print(a, b, c)

l = [3, 2, 1]
a, b, c = l  # unpacks the list l; prints "3 2 1"
print(a, b, c)

s = {3, 2, 1}
a, b, c = s  # unpacks the set s; prints "1 2 3" (set ordering)
print(a, b, c)

d = {'c': 3, 'b': 2, 'a': 1}
a, b, c = d            # unpacks the keys of the dict d; prints "c b a"
print(a, b, c)
ak, bk, ck = d.keys()  # unpacks the keys of the dict d; prints "c b a"
print(ak, bk, ck)
a, b, c = d.values()   # unpacks the values of the dict d; prints "3 2 1"
print(a, b, c)
(ak, a), (bk, b), (ck, c) = d.items()  # unpacks key-value tuples of the dict d
print(ak, bk, ck)      # prints "c b a"
print(a, b, c)         # prints "3 2 1"
```

The asterisk (`*`) can be used as an unpacking operator:

```
l = [3, 2, 1]     # a list
t = (4, 5, 6)     # a tuple
s = {9, 8, 7}     # a set
b = [*s, *t, *l]  # unpacks s, t, and l side to side in a new list
print(b)          # prints "[8, 9, 7, 4, 5, 6, 3, 2, 1]"
b = (*s, *t, *l)  # unpacks s, t, and l side to side in a new tuple
print(b)          # prints "(8, 9, 7, 4, 5, 6, 3, 2, 1)"
b = {*s, *t, *l}  # unpacks s, t, and l side to side in a new set
print(b)          # prints "{1, 2, 3, 4, 5, 6, 7, 8, 9}"
```

For dictionaries, we use the double-asterisks (`**`) or the (`zip`) function:

```
d1 = {'c': 3, 'b': 2, 'a': 1}
d2 = {'d': 4, 'e': 5, 'f': 6}
d = {**d1, **d2}
print(d)

keys = ['a', 'b', 'c']
values = [1, 2, 3]
print(type(zip(keys, values)))
d = dict(zip(keys, values))
print(d)
```

Functions

Python functions are defined using the `def` keyword. For example:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

You can also define variadic functions, like this:

```
def hello(*names, **kwargs):
    if 'loud' in kwargs and kwargs['loud']:
        print('HELLO, {}!'.format([name.upper() for name in names]))
    else:
        print('Hello, %s' % [name for name in names])

hello()                          # Prints "Hello, []"
hello('Bob', 'Fred')             # Prints "Hello, ['Bob', 'Fred']"
hello('Bob', 'Fred', loud=True)  # Prints "HELLO, ['BOB', 'FRED']!"
```

Classes

The syntax for defining classes in Python is straightforward and has the foolowing form:

```
class Greeter(object):
```

```
    # Static variable
    sneezes = 0

    # Constructor
    def __init__(self, name):
        # super().__init__()  # call to the constructor of the parent class
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

    @staticmethod
    def sneeze(n_a=1, n_o=2):
        print('A' * n_a + 'CH' + 'O' * n_o + '!!')
        Greeter.sneezes += 1

    def __str__(self):  # The str dunder (or magic) function
        return f'Greeter for {self.name}'

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
g.sneeze()           # Call a static method through an object; prints "ACHOO!!"
Greeter.sneeze()     # Call a static method through the class; prints "ACHOO!!"
print(g)             # Call __str__; prints "Greeter for Fred"
```

Python Technical Details. Middle

Convention about Variable Names

Convention	Example	Meaning
Single Leading Underscore	`_var`	This naming convention indicates a name is meant for internal use. Generally not enforced by the Python interpreter (except in wildcard imports) and meant as a hint to the programmer only. During import using the wildcard symbol (`from my_module import *`), these symbols will not be imported in the global namespace.
Single Trailing Underscore	`var_`	Used by convention to avoid naming conflicts with Python keywords.
Double Leading Underscore	`__var`	Triggers name mangling when used in a class context. Enforced by the Python interpreter to make name mangling
Double Leading and Trailing Underscore	`__var__`	Indicates special methods defined by the Python language. Avoid this naming scheme for your attributes.

Convention	Example	Meaning
Single Underscore	`_`	Sometimes used as a name for temporary or insignificant variables ("don't care"). Also, it is the result of the last expression in a Python REPL (interactive Python mode).

Inspect Python Objects

As we have already mentioned In Python, everything is an object.
The dir(obj) built-in function displays the attributes of an object. Attributes that all objects (typically) have:

- `__name__` - is the name of the object such as a function.
- `__doc__` - documentation string for the object.
- `__class__` - type name of the object.

Example:

```
a = 1
print(a.__class__)
```

Variables Introspection

- globals() - will give you a dictionary of global variables
- locals() and vars() - will give you a dictionary of local variables
- dir() - will give you the list of in-scope variables which includes locals, and enclosed variables from another function if you use the style in which you define a function inside the function.

Example:

```
#!/usr/bin/env python3

my_global = 1

def f():
  my_enclosing = 2

  def g():
    nonlocal my_enclosing
    global g

    my_local = my_enclosing

    print("locals:", locals())
    print("globals:", globals())
    print("in-scope:", dir())
  g()

f()

# OUTPUT
#   locals: {'my_local': 2, 'my_enclosing': 2}
#   globals: {'__name__': '__main__', ..., 'my_global': 1, ...}
#   in-scope: ['my_enclosing', 'my_local']
```

There are several different contexts/scopes in Python as has been described in Context in Python Section.

Global and Nonlocal Variables

In example Variables Introspection we have used nonlocal, and global keywords. You should use this variable access modifiers when you're going to write to the variable and provide an interpreter and hint about what you are going to do exactly:

- You want to define a new local variable and you provide the default value with the `=` operator
- You do not want to create a local variable, but instead, you want to write to a global variable or variable from a previous(nested) context.

global. The global statement is a declaration that holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. Global variable scope changed immediately to the module-level binding without any intermediate outer scope, relative to the current scope. Also, it is impossible to assign value to a global variable without using `global`. Although you can refer to global variables if you want to read from them.

nonlocal. The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. So nonlocal variables scope changed the scope to the outer function. This is important because the default behavior for binding is to search the local namespace first. Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

In C++/C#/Java, such nonlocal scope is impossible because in these languages you can not define a function inside another function. (except lambda)

Working with Tuples

A tuple consists of several values separated by commas. It is not possible to assign to the individual items of a tuple, however, it is possible to create tuples that contain mutable objects, such as lists. Tuples are immutable and usually contain a heterogeneous sequence of elements that are accessed via unpacking.

A tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). And so to create a tuple with one element is possible only with this required syntactic trick. Example:

```
a = (12,)
```

Empty tuples are constructed by an empty pair of parentheses. Example:

```
 a = ()
```

You can create a tuple from the list:

```
([1,2,23])
```

If you want to include tuples in another tuple you need to use parentheses `()`, so that nested tuples are interpreted correctly. Example of creating a tuple with 2 elements:

```
a=(1,(2,3,4,5,6))
```

Tuple packing, unpacking is an interesting idea at the language level.
https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists

Modules and Packages

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py`. Within a module, the module name (as a string) is available as the value of the global variable `__name__`.

To import the module the following instruction should be used:

```
import my_module
```

This does not add the names of the functions defined in `my_module` directly to the current namespace. Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. With this form of import statement all global symbols defined in `my_module` are available through `my_module.<function|variable name>`.

Next, there is a variant of the import statement that imports names from a module directly into the importing modules namespace:
`from my_module import func_f, func_g`

This statement does not introduce the module name in the global namespace of the current module.

The next variant is to import all names that a module defines `from my_module import *`, except variables and functions whose names are started with an underscore.

A module can contain function definitions.

But also it can contain executable statements. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

It is typically, but it is not required by interpreter design to place all import statements at the beginning of a module.

If the module name is followed by as, then the name following as is bound directly to the imported module. Example:

```
import my_module as my
```

To inspect the name defined in module `my` you should use the built-in function `dir()` e.g. in the following way `dir(my)`. It is used to find out which names a module defines.

In some sense, Python module files are files written with Python source code. When the Python module is executed as a script.

There are only two differences from code execution if comparing the launching script of the module and importing the module.

1. During launching the module the code in the module will be executed, just as if you imported it, but with the **name** set to "**main**". The part of the code for executing the script in the module typically has the following condition:

   ```
   if __name__ == "__main__":
       pass
   ```

   It provides means to distinguish behavior when the script is used for execution logic, rather than be a facade interface to some functionality.

2. When you launch Python scripts if all is OK in OS and you have enough privileges inside OS then the script will be launched and executed. However, the `import` statement (by default) is executed only once per Python session. See Also Module Reloading.

Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". Firstly you should have source code files with modules. To group these modules into the package you should place modules in one directory and create file `__init__.py` in this directory.

The `__init__.py` files are required to make Python treat directories containing the file as packages. This prevents directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute the initialization code for the package or set the `__all__` variable.

If execute code:

```
from package_name.module_name import *
```

Then `__all__` variables define the index of modules.

Reference between modules and packages

To use intra-package references imports use leading dots to indicate the current and parent packages involved in the relative import. Relative imports are based on the name of the current module.

A single dot means that the module or package referenced is in the same directory as the current location:

```
from . import echo
```

Two dots mean that it is in the parent directory of the current location—that is, the directory above:

```
from .. import formats
```

Rules for Search Modules and Packages

When you import a module via keyword import runtime importing the module via following the next rules:

1. Interpreter first searches for a built-in module. The built-in module names are built-in into the language and are listed in `sys.builtin_module_names`.
2. In a list of directories given by the variable `sys.path`.

Python programs can modify the `sys.path` variable itself, but the variable `sys.path` after starting the interpreter as a process Python Runtime initializes this variable with these locations:

- The directory containing the input script (or the current directory)
- Paths in environment variable PYTHONPATH
- The installation-dependent default such as the site-packages directory.

Comprehensions syntax

Comprehensions syntax provides a short syntax to create several iterable types.

List comprehensions in Python:

```
[expr(item) for an item in iterable if condition (item)]
```

set comprehensions in Python:

```
{expr(item) for an item in iterable if condition (item)}
```

Dictionary comprehensions in Python:

```
{someKey:someValue for someKey, someValue in someDict.items() if condition(someKey,
someValue)}
```

Generator comprehensions in Python:

```
(expr(item) for item in iterable if condition(item))
```

List comprehension is described in Python Tutorial [1]:
list comprehensions, nested list comprehensions

Random Interesting Construction

1. Work with the reverse sequence:

   ```
   for i in reversed ([1,3]):
       print i
   ```

2. Dictionary (also known as a map or associative array in other languages) can be built through `{}`, which takes as input list of pair-tuples with key and value separated by a column.

   ```
   emptyDictinoary = {}
   ```

3. Dict can be merged with another dict via update() method.

4. In Python another datatype which is built-in in the Language is set. On sets, you can perform set-theoretic operations on it. Example:

   ```
   aSet = {1,2}   # set
   ```

5. Creating an empty seta, possible only through the constructor because `{}` is reserved as an empty dictionary description and interpreter due to its design (it has a Dynamic Type System) can not derive information that you want an empty set, no dictionary. Example:

   ```
   emptySet = set()
   ```

6. Exceptions with variable names.

   See Handling Exceptions. Example:

```python
def hello(a):
  try:
    raise ValueError()
    return +2
  except (ValueError, TypeError)  as e :
    return -1
  finally:
    #clean code
    pass
```

7. There is a ternary expression similar to C/C++ `?:` expression. Example:

   ```
   sym = '0' if 1> 5 else ''1"
   ```

   Documentation: https://docs.python.org/2/reference/expressions.html#conditional-expressions

8. Launch the Python web server to share files from the current directory:

   ```
   python -m http.server
   ```

9. Deleted statement. Delete statement `del` is used to delete elements from a container such as a list. And also it is used to delete variables from the interpreter. After variable deletion, the variable is not defined. If a variable is not "defined" which means it has not been assigned a value or deleted, trying to use it for reading will give you an error. Trying to use it for write will create a new variable. Objects are never explicitly destroyed. 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.

10. Multiple assignments in Python:

    ```
    a, b = 0, 1
    ```

Technical Details about Functions

About Indentation

Unfortunately, Python was designed using indentation, and there are no scopes constructed with `{}` as in C/C++. During writing language construction sometimes it can be shorter, but sometimes when you 3-9 nested loops of Algorithm logic, the indentation makes the problem less readable and more error-prone. In principle, you can use both spaces and tabs, but it is recommended to use spaces instead of tabs according to this recommendation:
https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces

Arguments of the function and return value

The first statement of the function body can optionally be a string literal - this string literal is the functions documentation string, or docstring.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table.

Variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless you use `global` and `nolocal` statements). Although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called. Arguments are passed by value, where the value is always an object reference, not the value of the object. A function definition associates the function name with the function object in the current symbol table.

The `return` statement returns with a value from a function. Return without an expression argument returns None. Falling off the end of a function also returns `None`.

Default Value of Argument for a function

The most useful form is to specify a default value for one or more arguments. Important warning: The default value is evaluated only once.

Keyword and Positional Arguments

Functions can be called using keyword arguments of the form `kwarg = value`. Keyword parameters are also referred to as *named parameters*. Rules for using keyword parameters are the following:

- In a function call, keyword arguments must follow positional arguments
- All the keyword arguments passed must match one of the arguments accepted by the function
- No argument may receive a value more than once

Syntax to split positional and keyword

```
def f(pos_only, pos_only_2, /, pos_or_kwd, *, kwd_only_1, kwd_only_2):
    pass
```

Optionally used symbols `/` and `*` indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only.

Varying Number of Arguments

You can specify that a function can be called with an arbitrary number of arguments.
One way is to use syntax to wrap varying positional arguments in a tuple. Example:

```
def f(a, *args):
    print(type(args))
```

The second way is to use syntax to wrap varying keyword arguments into a dictionary. Specifically, when a final formal parameter in the function is in the `**name` it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. Example:

```
def f(a, **args):
    print(type(args))
```

Next, if you want to call a function by passing a *tuple* and *dictionary*, but transfer this not as objects but make a substitution of the content of *tuple* and *dictionary* as a formal argument you should:

- Expand list and tuple with `*`
- Expand the dictionary (dict) with `**`

Example:

```
def printInfo(*x, **kv):
   print(x)
   print(kv)

x = (1,2)
kv = {'a':2,'b':4,'c':6}

printInfo(*x, **kv)
printInfo(*x, aa = 1, bb = 9, cc = 1)

# Output:
# (1, 2)
# {'a': 2, 'b': 4, 'c': 6}
# (1, 2)
# {'aa': 1, 'bb': 9, 'cc': 1}
```

Small anonymous functions can be created with the lambda keyword

Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression.

Function and Type Annotation

Function annotations (link) and type annotation (link) are completely optional metadata information about the types used by user-defined functions (see PEP 3107 and PEP 484 for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary.

Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, similar to the trailing return type in C++.

Annotations do not affect any part of the function (if the function does not use this system meta information in its logic).

General Convention is using parameter annotations with expressions that produce a `type` value, and this functionality is used to augment Python with type information. Example:

```
def f(ham:str , eggs:str  =  'eggs' )->str:
    print( "Annotations from function:" , f . __annotations__ )
    return "1"

print( "Annotations outside function:" , f . __annotations__ )

f("1", "2")
```

Function Decorators

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in a nested fashion.

More information is available in [2]:
https://docs.python.org/3/reference/compound_stmts.html#grammar-token-python-grammar-decorators.

For example, the following code will measure the time for function execution (it has been taken from https://stackoverflow.com/questions/5478351/python-time-measure-function):

```
#!/usr/bin/env python3
import time

def timing (f):
  def wrap(*args):
      time1 = time.time()
      ret = f(*args)
      time2 = time.time()

      print ('{} function took {:.3f} ms'.format(f.__name__, (time2-time1) * 1000.0))

      return ret

  return wrap

# Use Attributes
@timing
def f1(seconds):
  time.sleep(seconds)

# Similar code without attributes
def f2(seconds):
  time.sleep(seconds)

def f2_with_time(seconds):
  funcion_to_call = timing(f2)
  funcion_to_call(seconds)

f1(1.5)
f2_with_time(1.5)
```

This concept is called decorator because the intention of using it is to decorate function calls with specific pre/post-processing.

Classes in Python

Classes provide a means of bundling data and functionality together. In C++ normally terminology class members, the data members are public, and all member functions are virtual.

The method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. Unlike C++ built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax can be redefined for class instances.

The super() lets you avoid referring to the base class explicitly, which can be nice sometimes.

```
#!/usr/bin/env python3

class Base:
  def f(self): print("Base")
  def fproxy(self): self.f()

class DerivedA(Base):
  def f(self): print("DerivedA")
```

```
class DerivedB(Base):
  def f(self): print("DerivedB")


class DerivedAB(DerivedA, DerivedB):
  def f(self):
      print("DerivedAB")
      Base.f(self)                  # Call Base explicitly
      super(DerivedAB, self).f()    # Call one of direct base with using super()
      DerivedB.f(self)              # Call DerivedB explicitly

obj = DerivedAB()
obj.fproxy()
```

Python Technical Details. Advanced.

Special or Magic method for classes

Magic is an official term used by the Python community, even though in professional literature this term is used rarely. This informal name shines a light that a lot of things inside the Python community happen informally without any standardization. The effect that both styles (formal and informal) can coexist can be obtained into look into API and development style for Android OS and Linux/Windows OS. The development for Android OS is mostly cowboy style.

A class can implement certain operations that are invoked by special syntax.
A complete list of these special methods is available in The Python Language Reference [2] https://docs.python.org/3/reference/datamodel.html#special-method-names.

Module Reloading

For efficiency reasons, each module is only imported once per interpreter session.
If you change your modules source code you have two options on how to reload this module:

- Restart the interpreter
- Use reload() function from imp module which provides access the import internals. Example:

```
import imp
imp.reload(module name)
```

Encoding for Files

The text file is iterable and the content of the file is returned line by line. You can check the used default encoding for reading text files:

```
import sys
print(sys.getdefaultencoding())
```

If you want to open the file in the specified encoding you can specify this in open().

To open and close files you can use open()/close() calls. One shorthand provides:

- The opening of the file
- Closing in success/exception
  is with statement.

Example:

```
with open("my_file.txt", "rt") as f:
  for line in f:
      print(line)
```

The reason for closing files is in that some objects contain references to external resources such as open files in the filesystem. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually with a `close()` method.

Defaultdict

Defaultdict is a subclass (derived class) of the built-in dict class. It can be found in the collections module in Python standard library.

Defaultdict is working mostly like a std::map in C++. When the key is encountered for the first time and it is not already in the mapping then an entry value corresponding to the requested *key* is automatically created using the default_factory function and is instantiated.

Example:

```
from collections import defaultdict

m1 = defaultdict(int)
# default value will be an integer value of 0
m2 = defaultdict(str)
# default value will be an empty string ""
m3 = defaultdict(list)
# default value will be an empty list []
```

Match Statement

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is similar to a switch statement in C/C++ conceptually.

```
match status:
       case 400:
           return "Bad Request"
       case 404:
           return "Not found"
       case 418:
           return "I'm a teapot"
       case _:
           return "Something's wrong with the internet"
```

The variable name `_` acts as a wildcard and never fails to match, similar to `default:` in C/C++.

Walrus

In Python, unlike C and C++, assignment inside expressions which is used in the conditional statement must be done explicitly with the walrus operator `:=`.

The operator `:=` in Python can be used in the context when you want to perform an assignment inside an expression that is used for a condition.

It's the same as the operator `=` in C++ but with restricted context.

```
a=2
if a:=1:
  print(a)

# Output
# 1
```

Generators

The class-based iterators is a functionality that is implemented via definition:

- `__iter__` method
- `__next__` method

See also Interface and Protocols in this document. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Generators in python are a specific type of functions that uses the (`yield`) statement. Example:

```
def one_two_three():
    x = 3
    yield x // 3
    x -= 2
    yield x + 1
    yield x * 3

for i in one_two_three():
    print(i)  # prints 1, 2, 3 in sperate lines
```

The key feature of **generator** is that the local variables and execution state are automatically saved between calls. When generators terminate, they automatically raise StopIteration exception.

The generators functionality dies not return a value via return statement, instead, they send a value via yield statement. See also: link to generators.

```
def gen():
    print ("Entry point in gen") # This will be executed only once, even a function
yields several values
    yield 1 # emit value
    yield 2 # emit one more value

a = gen ()
print(next(a))   # use it to explicitly first yeild statement
print(next(a))   # run to second yield statement, etc.

# Next yield will throw a StopIteration exception
```

Conda/Pip/Venv package and environment managers

Package managers

Conda has two goals:

1. Conda is a package manager.
2. Conda is an environment manager.

Each project can have a specific requirement in a specific version of the library and this is the motivation behind the environment manager idea.

To install Conda there are two ways to:

1. Anaconda distribution contains Conda and other things.
2. Install Miniconda (https://docs.conda.io/en/latest/miniconda.html)

I prefer not to use pip directly, but instead use command for call pip directly from Python:

```
python -m pip list
```

It's useful to eliminate problems with various installations of Python interpreters.

Package manager commands comparison:

#	Command in Conda	Command in Pip	Description
1	conda search	pip search	Search package
2	conda install/uninstall	pip install/uninstall name	install/uninstall package
3	conda update	pip install --upgrade name	upgrade package
4	conda install package=version	pip install package=version	Install a specific version of package
5	conda list	pip list	List of installed packages
6	conda install mnist --channel conda-forge	pip install somepkg --extra-index-url http://myindex.org	Install package from non-standart place
7	conda update package_name	pip install -U package_name	Upgrade package
8	Use pip for it, not conda	pip install *.whl	Install package from local WHL (Python wheel packaging standard) distribution.
9	conda list | grep <package_name>	pip show <package_name>	Show information about package

Conda cheatsheet:
https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf

Conda documentation by the tasks:
https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/index.html

How to use pip:
https://pip.readthedocs.org/en/stable/

Python Package Index repository:
https://pypi.python.org/pypi - standard packages repository

Environment managers

The venv module is a standard virtual environment in Python. In this section, we will compare venv with conda. If you want to obtain a list of available conda environments please use conda info -e.

#	Command in Conda	Command in venv module	Description
1	conda create --name my	python -m venv my	Create an environment with the name "my"
2	conda create --name my python==3.6.4 scikit-learn	pip install	Create an environment with several packages in it
3	conda activate my	source ~/envs/my/bin/activate	Activate specific environment
4	conda deactivate	source ~/envs/my/bin/deactivate	Deactivate environment
5	conda list	pip freeze	Get List of installed packages in the current environment
6	conda env export --file myenv.yml	pip freeze > requirements.txt	Export information about packages in the current environment
7	conda env create --file myenv.yml	pip install -r requirements.txt	Install all packages from requirement list
8	conda remove --name myenv --all	Remove directory in filesystem with environment and it's scripts	Remove environment

Python Notebooks

Jupyter documentation is available here https://docs.jupyter.org/en/latest/. Jupyter Notebooks presents a way to create source code mixed with documentation in the form of Markdown. Also, Notebooks can serialize the results of numerical computation in the form of graphics into a single file.

Python Notebooks by itself does not provide means to debug Python Notebooks, however some IDE as Visual Studio Code support debugging of code inside Python Notebooks.

The following command will start the web server and open web-client:

```
python -m notebook
```

Start the web server and allow connection from outside (incoming connection from all network interfaces will be accepted):

```
python -m notebook --ip 0.0.0.0 --port 8888
```

> If you have problems with accessing the web server the first thing is to take a look if the port that you have specified listens to the port for incoming network connection. In Posix OS it can be done by using the netstat command:
>
> ```
> netstat -nap | grep -E "tcp(.)*LISTEN"
> ```

> Please be aware that if the server is running on some machine it can be a case that the computer administrator blocks the ports. In Linux-based OS, by default, in most cases, people use iptables to configure acceptance of incoming connections.
> The first thing you can try to execute from the root is the following command to allow all incoming connections from all interfaces:
>
> ```
> iptables -I INPUT -j ACCEPT
> ```

Python notebook files extension is `*.ipynb`. The notebook contains code snippets and markdown text organized by cells. Cells roughly speaking can be of two types:

- Code
- Markdown

Working in a web-based interface.

Command	Description
ALT+Enter	Append cell (something like the section in Matlab)
a_1+a_2	It's possible to use Latex in Markdown
\%matplotlib nbagg	Append separate widget where output should be plotted
\%matplotlib inline	Inline output into the notebook
Left click on cell	Select to edit
Esc	Deselect for edit
Shift + down / Shift + up	Append cell to selection
Shift + Enter	Execute (selected) cell and go to next cell
\%timeit	Built-in command in Jupyter does measure the time of command. Documentation.
CTRL+M, B	Create code cell
CTRL+M, M	Convert code cell to text cell
CTRL+M, Y	Convert text cell to code cell
%%bash; ls -1; lsb_release --all	Magic sequence to execute bash commands via `%%bash`. Documentation.
!pip install numpy	Magic command for executing bash command at code cell via using `!` mark. Documentation.

Example of usage timeit inside code cells:

```
%timeit for _ in range(10**3): True
```

Documentation:
https://ipython.org/ipython-doc/dev/interactive/magics.html

PyTorch resources

PyTorch is a big numerical package.

Description	Link
PyTorch index for several topics	https://pytorch.org/docs/stable/index.html
The PyTorch Cheat Sheet	https://pytorch.org/tutorials/beginner/ptcheat.html
PyTorch API	https://pytorch.org/docs/stable/torch.html
Tensors	https://pytorch.org/docs/stable/tensors.html
Parameters	https://pytorch.org/docs/stable/generated/torch.nn.parameter.Parameter.html
Tensors required grad modification	https://pytorch.org/docs/stable/tensors.html#torch.Tensor.requires_grad_
Moving tensor to GPU	https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html
Data relative/Data loader	https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader
Data relative/Dataset	https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset
Data relative/Custom data loader	https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
Module/About Modules	https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
ModuleBase class for all neural network Module/Layer	https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module
The module which allows building a network sequentially	https://pytorch.org/docs/master/nn.html#torch.nn.Sequential
Turn on/off module training mode	https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module.train
Linear Layer/Module	https://pytorch.org/docs/master/generated/torch.nn.Linear.html#torch.nn.Linear
Relu layer	https://pytorch.org/docs/master/generated/torch.nn.ReLU.html
Data Transform	https://pytorch.org/vision/stable/transforms.html

Description	Link
Image/Tensor transformer	https://pytorch.org/vision/0.8/transforms.html#torchvision.transforms.ToTensor
Tensor values normalization	https://pytorch.org/vision/0.8/transforms.html#torchvision.transforms.Normalize
Losses/Binary Cross-Entropy loss	https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html
Losses/Binary Coss-Entropy with logits loss	https://pytorch.org/docs/master/generated/torch.nn.BCEWithLogitsLoss.html
Losses/Cross-Entropy loss	https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
Optimizers	https://pytorch.org/docs/stable/optim.html
Autograd	https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

|

> Comment: In the context of deep learning the logits mean the layer or scalars from R that are fed into softmax or similar layer in which the image(or output) is a probabilistic simplex.

Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to MATLAB. Documentation: http://matplotlib.org/api/pyplot_api.html

Plots

The most important function in matplotlib is `plot`, which allows you to plot 2D data. Here is a simple example:

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0 , 10 , 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.grid(True)
plt.show()
```

With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
```

```
# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

SubPlots

You can plot different things in the same figure using the `subplot` function. Here is an example:

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

# Compute the x and y coordinates for points on sine and cosine curves.
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has 2 rows and 1 column.
figure, axes = plt.subplots(2, 1)

# Make the first plot
ax = axes[0]  # plt.subplot(2, 1, 1)
ax.plot(x, y_sin)
ax.set_title('Sine')

# Set the second subplot as active, and make the second plot.
ax = axes[1]  # plt.subplot(2, 1, 2)
ax.plot(x, y_cos)
ax.grid(True)
ax.set_title('Cosine')

# Show the figure.
plt.show(figure)
```

Show the image with Matplotlib

You can show image with the following code snippet:

```
import sys
import matplotlib.pyplot as plt
import numpy as np
img = np.random.randint(low =  0, high =  255, size =  (16 ,  16,  3))
plt.imshow(img)
plt.show()
```

Show sub-images:

```
import sys
import matplotlib.pyplot as plt
import numpy as np
out_arr = np.random.randint(low =  0,
                            high = 255,
```

```
                              size = (16, 16, 3))
plt.figure(figsize = (15 , 15))

for n in range(16):
  ax = plt.subplot( 4 , 5, n + 1 )
  ax.title.set_text( 'Plot #'  +  str ( n + 1 ) )
  plt.imshow ( out_arr )
  plt.axis( 'off' )
  plt.subplots_adjust (wspace = 0.01 , hspace = 0.1 )

plt.show( )
```

Cython

Cython is Python with C data types. Official documentation: https://cython.readthedocs.io/en/latest/index.html.

Cython has two major use cases:

1. Extending the CPython interpreter with fast binary modules.
2. Interfacing Python code with external C libraries.

Almost any piece of Python code is also valid Cython code. Usually, the speedups are between **2x** to **1000x**. It depends on how much you use the Python interpreter compared to situations where your code is in C/C++ Libraries.

There are two file types for Cython:

- `*.pyx` - something similar to C/C++ source files.
- `*.pxd` - something similar to C-header files. PXD files are imported into PYX files with the `cimport` keyword.

The files in Cython have several functions inside them:

- `def-functions` - Python functions are defined using the def statement, as in Python. Only Python functions can be called from this function.
- `cdef-functions` - C functions are defined using the new compare to Python `cdef` statement. Within a Cython module, Python functions and C functions can call each other.
- `cpdef-functions` - is a hybrid of `cdef` and `def`. It uses the faster C calling conventions when being called from other Cython code and uses a Python interpreter when they are called from Python. Essentially it will create a C function and a wrapper for Python.

During passing argument from `cdef` to `def` functions and vice versa there is an automatic type casting is occurring:
https://cython.readthedocs.io/en/latest/src/userguide/language_basics.html#automatic-type-conversions

How to optimize Python Code with Cython

0. Install Cython for your Python interpreter: `python -m pip install Cython`.
1. First step is to take the usual Python file "*.py" and change the extension to* ".pyx".
2. Second and most powerful step that can be used now during using Cython you can append type information to variables. In practice especially if you are using loops it brings good speedup immediately. Examples:

```
#!/usr/bin/env python3

cdef int x,y,z
cdef char *s
cdef float f1 = 5.2    # single precision
cdef double f2 = 40.5  # double precision
cdef list languages
cdef dict abc_dict
cdef object thing
```

3. Next step you can append extra modifiers to functions. Example:

```
#!/usr/bin/env python3

# Can not be called from Python
cdef sumCDef():
  cdef int i
  cdef int s = 0
  for i in range(10):
    s += i
  return s

# Can be called from Python and Cython
def sumDef():
  cdef int i
  cdef int s = 0
  for i in range(10):
    s += i
  return s

# Can be called from Python and Cython effectively
cpdef sumCpDef():
  cdef int i
  cdef int s = 0
  for i in range(10):
    s += i
  return s
```

4. Next Step is to build your code because Cython is not an interpretable Python language extension. The script that describes that you want to build all *.pyx files in the current directory, which is typically used for projects that use Cython:

```
#!/usr/bin/env python
# filename: setup.py
# pip install Cython
from setuptools import setup
from Cython.Build import cythonize
setup(
    name        = 'Test',
    ext_modules = cythonize("*.pyx"),
    zip_safe    = False,
)
```

5. Launch the build process from the previous description:

```
python build.py build_ext --inplace
```

The output of this command is

- `*.so` in Unix-like OS.
- `*.pyd` in Windows.

6. If the build process was successful then to import your module into the Python interpreter you can use the usual `import` statement:

```
import my_module
```

About Cython Language

The `cdef` statement is used to declare C variables, either in the local function scope or in the module level:

```
cdef int i, j, k
```

With cdef, you can declare `struct`, `union`, `enum`
(See Cython documentation)
:

```
cdef struct Grail:
  int age
  float volume
```

You can use `ctypedef` as an equivalent for C/C++ typedef:

```
ctypedef unsigned long ULong
```

It's possible to declare functions with cdef, making them C functions:

```
cdef int eggs(unsigned long l, float f)
```

References with further details:

- https://cython.readthedocs.io/en/latest/index.html
- https://pythonprogramming.net/introduction-and-basics-cython-tutorial

Easy Interoperability with Standar C Library

```
#!/usr/bin/env python3

cdef extern from "math.h":
    double sin(double x)

from libc.math cimport sin
from cpython.version cimport PY_VERSION_HEX

cpdef double f(double x):
  print("CPython version for which code is compiled:",
        PY_VERSION_HEX)
  return sin(x * x)
```

External declaration in form `cdef extern from "math.h":    double sin(double x)` declares the `sin()` function in a way that makes it available to Cython code. It instructs Cython to generate C code that includes the `math.h` header file.

The C compiler will see the original declaration in math.h at compile time. Importantly Cython does not parse "math.h" and requires a separate definition in such form.

It is possible to declare and call into any C library as long as the module that Cython generates is properly linked against the shared or static library.

Example of function Integration in Cython and Python

```
# Source File: integration.pyx
# Depencies:
#  python -m pip install cython

import time

def timing(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        print('{:s} function took {:.3f} ms'.format(f.__name__, (time2-time1)*1000.0))
        return ret
    return wrap

def f(double x):
    return x ** 2 - x

@timing
def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx

    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

@timing
def integrate_f_std(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

# Build with:
#   python setup.py build_ext --inplace
#
# Where setup.py:
#  from setuptools import setup
#  from Cython.Build import cythonize
#  setup(
#     name        = 'Reduction Test',
#      ext_modules = cythonize("*.pyx"),
#      zip_safe    = False,
#    )

# Use from Python after build:
#   import integration
#   integration.intergrate_f(0.0,100.0,1000)
```

```
#   integration.integrate_f_std(0.0,100.0,1000)
```

Numpy

Numpy is the core library for scientific computing in Python.
It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

Check out the numpy reference (http://docs.scipy.org/doc/numpy/reference/) to find out much more about numpy beyond what is described below. You can find the full list of mathematical functions provided by numpy in: http://docs.scipy.org/doc/numpy/reference/routines.math.html.

Numpy provides various functions for manipulating arrays; you can see the full list in: http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html.

Broadcasting explanation: https://numpy.org/doc/stable/user/basics.broadcasting.html.

(If you think that Broadcasting is incorrect thing to be designed in your first place - you are not alone).

To use `numpy` library in your project you need to:

- Install it with your package manager via `pip install numpy`
- Import numpy via `import numpy as np`

Arrays

A numpy array is a grid of values, **all of the same type**, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array. The shape of an array is a tuple of integers giving the size of the array along each dimension.

- Rank of array - is number of dimensions.
- Tensor (in Machine Learning / Deep Learning) - is a name used to describe multidimensional arrays.
- Shape - description of dimensions for multidimensional array organized as a tuple. Each dimension of multi dimensional array is called as "dimension" or "axe".

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy provides many functions to create arrays:

```
a = np.zeros((2,2))   # Create an array of all zeros with shape 2x2
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
```

```
                      #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.42  0.42]
                             #               [ 0.42  0.42]]"

f = np.arange(5)             # Create an array with the values from 0 to 4
print(f)                     # Prints "[0 1 2 3 4]"

g = np.arange(4.2, 7.1, 0.5)  # Create an array with the values from 4.2 to
                              # 7.1 on steps of 0.5
print(g)                      # Prints "[4.2 4.7 5.2 5.7 6.2 6.7]"

h = np.linspace(10, 20, 5)  # Create an array with 5 equally spaced values
                            # between 10 and 20
print(h)                    # Prints "[[10. 12.5 15. 17.5 20.]"
```

Array indexing

Numpy offers several ways to index into arrays. Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array. Example:

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4],
              [5,6,7,8],
              [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2 (and not 3); b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.

print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
print(b)
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array:

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4],
              [5,6,7,8],
```

```
              [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.

# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]

print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"

print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

The ellipsis is used in NumPy to slice higher-dimensional data structures. It's designed to mean at this point, insert as many full slices (`:`) to extend the multi-dimensional slice to all dimensions.

```
a = np.array ([[1,2,3], [4,5,6]])
b = a [...]
b[0,0]=11
print(a == b)
```

Also you can put, the new axis is used to increase the dimension of the existing array by one more dimension when used once.

```
import numpy as np
a = np.array ([[1,2,3], [4,5,6]])
b = a [..., np.newaxis]
print(b.shapea)
```

Boolean array indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
```

```
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
x = np.array([1, 2])       # Let numpy choose the datatype
print(x.dtype)             # Prints "int64"
x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
print(x**0.5)
```

Important notice about syntax for Matrix Multiplication

Note that unlike MATLAB, `*` is elementwise multiplication, not matrix multiplication.

In numpy instead of use `*` you need to use the `dot` function or operator `@` to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. `dot` is available both as a function in the numpy module and as an instance method of array objects:

```
x = np.array([[1,2],
              [3,4]])
y = np.array([[5,6],
              [7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors
print(v.dot(w))
print(np.dot(v, w))
print(v @ w)

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x@y)
```

Various Utility functions in Numpy

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))          # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix. To transpose a matrix, simply use the `T` attribute of an array object:

```
x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

For reshaping matrix into different form you can use reshaping function `np.reshape`:

```
x = np.arange(12)
y = x.reshape((3,4))

print(x)    # Prints "[ 0  1  2  3  4  5  6  7  8  9 10 11]"
print(y)    # Prints "[[ 0  1  2  3]
            #          [ 4  5  6  7]
            #          [ 8  9 10 11]]"
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array. Unfortunately in practice that mechanism can lead to confusion. So be very careful!

Example: add a constant vector to each row of a matrix.

```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])

#============================================================================
# Add the vector v to each row of the matrix x with an explicit loop
#============================================================================
y = np.empty_like(x)   # Create an empty matrix with the same shape as x
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)

#============================================================================
# Add the vector with tiling
#============================================================================

vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
```

```
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"


#==========================================================================
# Add vector with broadcasting
#==========================================================================
# Numpy broadcasting allows us to perform computation without actually creating
multiple copies of **`v`**.
# Consider this version, using broadcasting.
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y

v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

References

Introduction document

[1] Python Tutorial: https://docs.python.org/3/tutorial/

Reference official materials

[2] Python Language Reference: https://docs.python.org/3.8/reference/index.html

[3] Built-in Types: https://docs.python.org/3/library/stdtypes.html

[4] Table of content (index) for Python: https://docs.python.org/3/contents.html

[5] Python Enhancement Proposals: https://www.python.org/dev/peps/

[6] Description of the meaning of various special member functions: https://docs.python.org/3/reference/datamodel.html#emulating-callable-objects

[7] Python standard library: https://docs.python.org/3/library/index.html#library-index

Mapping concepts from other languages/libraries to Python language/libraries

[8] Matlab/Numpy translation: http://mathesaurus.sourceforge.net/matlab-numpy.html

Tutorial for Libraries

[9] http://cs231n.github.io/python-numpy-tutorial/

Howto

[10] http://www.java2s.com/Tutorial/Python/0400__XML/AccessingChildNodes.htm

[11] http://book.pythontips.com/

[12] How to for Python: https://docs.python.org/3/howto/index.html

Repositories

[13] Find, install, and publish Python packages:
https://pypi.org/