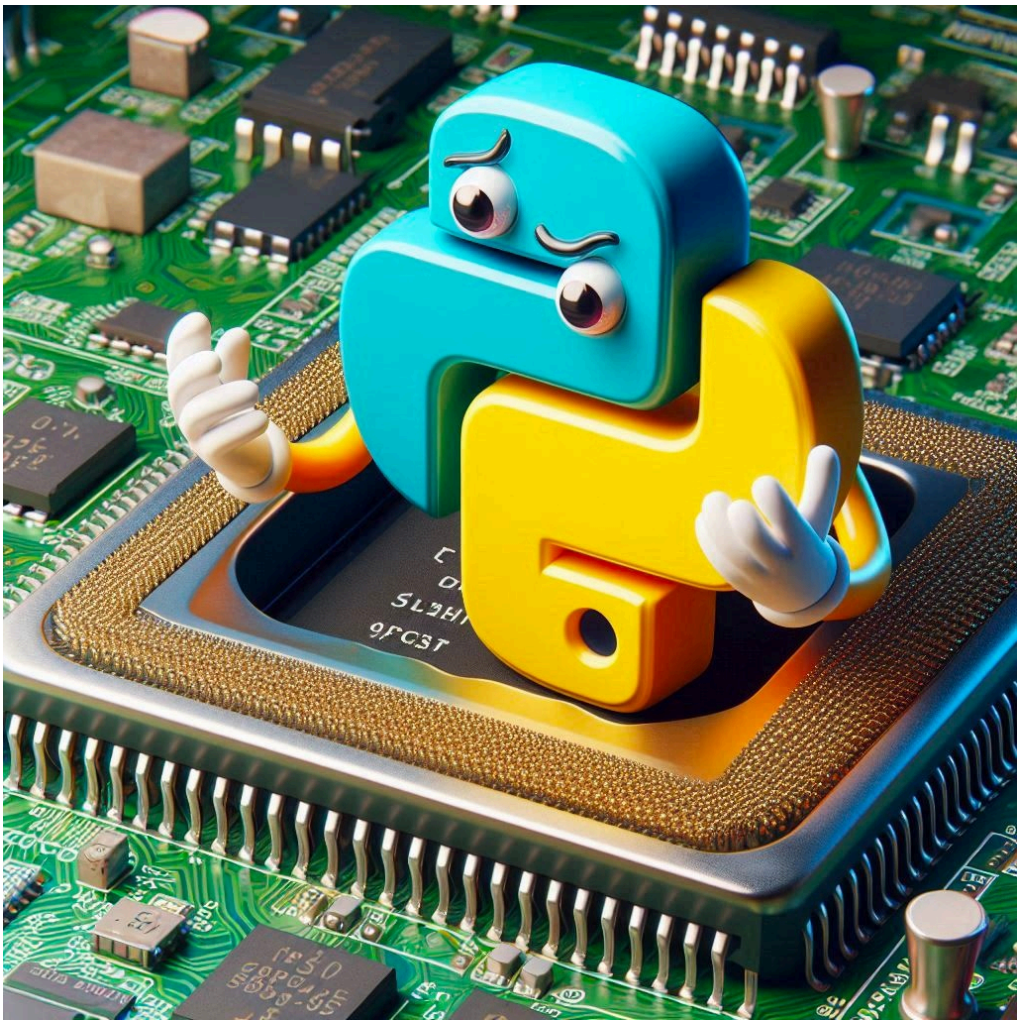
Technical Note. Exploring Python3 Language from a Computing Perspective



[Konstantin Burlachenko]

Revision Update: Apr 2, 2025

Revision Version: [Release v2.0]

Introduction

If you search "Python Tutorial" on Google, you will get about 695 million results. Among them, one tutorial stands out as a reliable and authoritative source - it is the official tutorial of the Python programming language, written by its creator Guido Van Rossum (https://en.wikipedia.org/wiki/Guido_van_Rossum). You can find it here: https://docs.python.org/3/tutorial/index.html. This official Python tutorial (https://docs.python.org/3/tutorial/index.html) is based on the most widely used Python distribution, CPython, which you can download from https://www.python.org/. There are also other Python distributions, such as Jython and Python for .NET, but they are less popular.

Python is a popular scripting language. It is very well-designed in style and has been picked up seriously by a broad category of people. However, to be honest, we should not forget that it is a scripting language. And it has a lot of technical limitations common to all scripting languages described in the section of this document C++ Technical Note / Downsides of Interpretable Languages. In addition, it lacks an official ISO/IEEE standard. The lack of standardization has its pros. and cons., which we will not touch in depth.

You can check the popularity of Python scripting language in this world from these two sources:

- The TIOBE-Index ranks programming languages by their popularity on their metrics.
- The Google-Trends, which shows the relative interest in different topics over time in society.

With this note, the authors have three objectives:

1. To explain some technical details about certain language features from Python 3.11 with examples. This can be a useful resource for those who found the official Python Tutorial too complex or confusing. This can also help to fill in the gaps in one's understanding of the subject.
2. Give a brief overview of various tools that can assist development in Python if you are using this Language. These tools are mainly developed to debug and profile executable programs inside Windows and Linux OS families.
3. Provide a bridge between Python and System-Aware mentality, which will reveal more information about the consumed resources of your scripts.

> If, after reading this note, you wish to contribute to this note, please take a look into the contributions section.

Motivation from Hello World Measurements

Consider the "Hello World" application implemented in C++23:

```
// Compiled with Microsoft Visual Studio Community 2022 (64-bit) Version 17.11.4;
Compiler Flags: /O2 /GL /std:c++latest
#include <print>
#include <stdio.h>
int main() {
    std::println("Hello World!");
    getchar(); // Wait for IO from standard input to not kill process by OS and
have the ability to inspect process
    return 0;
}
```

Now, let's create the equivalent "Hello World" script in Python:

```
#!/usr/bin/env python3
# Launched with: Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC
v.1927 64 bit (AMD64)] on win32

print("Hello World!")
input()    # Wait for IO from standard input to not kill process by OS and have
the ability to inspect process
```

With the tools described in this document, you can precisely measure the system resources each program consumes on both Windows and Linux operating systems and go beyond considering only line numbers (7 lines in the C++ program and 2 lines in the Python script).

Below is a sample comparison of resource usage on Windows 11 Home, highlighting key differences in performance and resource allocation between C++ and Python:

Resource Metric	C++ Application for Hello World	Python Script for Hello World
Consumed Clocks of CPU	6M clocks	74M clocks
Peak Private Bytes	0.4 MB	3.6 MB
Page Faults	970	2531

Resource Metric	C++ Application for Hello World	Python Script for Hello World
Peak Working Set	3.7 MB	9.8 MB
Peak OS Handles	43	82
File I/O Bytes	0	163 348
File I/O Operations	5	635
Registry Operations	52	154
Extra Runtime Environment (not part of OS)	0	1.60 GB for Python 3.9 (MSC v.1927 64 bit (AMD64))
Executable Script/Application	112 KB	274 bytes

This comparison made through SysInternals Suite from Mark Rusinovich et al highlights the considerable resource disparities between a low-level, compiled language (C++) and an interpreted, high-level scripting language (Python).

By investigating such basic applications, we can gain insight into the underlying costs associated with higher-level language abstractions and runtime environments, guiding informed choices for resource-sensitive applications.

What is Python

Python is an interpretable scripting language. Historically, Python was designed originally only as a replacement for Bash, which has been described in that Blog Post written by the original author back in 2009:

> "...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.

As we have already stated, the most common implementation of the Python interpreter is CPython. It is named CPython because its core was written in C. Unlike most programming languages, Python does not have an official standard. The only available resources for people who want to create their own Python interpreter or create some system software:

- Language reference: https://docs.python.org/3/reference/
- Source code of Python interpreter: https://github.com/python/cpython

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 which are not the same as instructions for CPU). Even though Python is interpreted, internally, the commands are translated into Python Virtual Machine before execution. If you want to see what the commands look like, you can use the dis module to help you with that:

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

import dis, sys

def f(x):
    xmy = xmy + 1
    return xmy

print("="*50)
print("Python version: ", str(sys.version).replace('\n', ' '))
print("="*50)
dis.dis(f, file=sys.stdout)
```

Output:

```
==================================================
Python version:  3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927
64 bit (AMD64)]
==================================================
  6           0 LOAD_FAST                1 (xmy)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 STORE_FAST               1 (xmy)

  7           8 LOAD_FAST                1 (xmy)
             10 RETURN_VALUE
```

Documentation:

- dis - Disassembler for Python bytecode
- opcode - Python Bytecode Instructions

Where to Learn About Python Officially

There are several resources from which anyone can learn about Python scripting language:

1. Firstly, there is a book written by Guido van Rossum (https://en.wikipedia.org/wiki/Guido_van_Rossum), who is the original author of this language. Today it has been converted into a pretty big tutorial, and it is not named as a "book", but named as a "tutorial": https://docs.python.org/3/tutorial/
2. All compiled documentation is available here in various formats (including PDF and HTML) : https://docs.python.org/3.11/download.html. The archive contains all the documents for the Python version that you will select.
3. Python Language Reference: https://docs.python.org/3.8/reference/index.html. For example, this is a link to the detailed description of different built-in functions for user-defined classes that you need to implement in your classes to support some built-in operators.
4. Python Interpreter is distributed with various standard modules called Python Standard Library: https://docs.python.org/3/library/index.html. You can find documentation that describes Standard Types through this link.

5. The CPython interpreter website contains a full-text search of Python documentation. You can use this utility both in terms of learning and in terms of finding details if you already know this scripting language: https://docs.python.org/3/search.html
6. The syntax of most Programming Languages is typically described with Backus Naur forms for Context-Free-Grammars (CFG). These grammar rules (which describe syntax rules) can be found here (in case you need to go into the level of specific construction rules): https://docs.python.org/3/reference/grammar.html
7. The Python scripting language (and another scripting language, such as Bash (https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html)) contains plenty of jargon and specific terminology. The Python glossary can be found here: https://docs.python.org/3/glossary.html
8. If you need to understand how some built-in type is implemented, you more likely need to go to the level of a source code of Python interpreter: https://github.com/python/cpython
9. A class can implement certain operations that can be invoked by special syntax. A complete list of these special methods is available in the Python Language Reference: https://docs.python.org/3/reference/datamodel.html#special-method-names.

Finally, some extra materials can be found in the references section of this document.

Backgrounds

How Typical Compute Device is Working

It's important at least to understand that computational devices do not execute code in Java, C#, Matlab, or Python. Compute devices (such as Central Processing Units and Graphical Processing Units) can not do it. The real computation devices execute binary code compiled in the form of Instruction Set Architecture(ISA), which connects software implementation with hardware implementation of electrical circuits that represent ISA functionality.

A simplified CPU device has computation cores that execute arithmetic operations via reading arguments from memory or registers and writing results back to memory or registers. Virtual memory provides the scratch storage of input and output results for your process (and other processes running in the Operating System(OS)).

The control unit in the CPU controls the execution and operation of Electrical Components (developed by Electrical Engineers). Finer details about how CPU works in details can be obtained from *System Architecture*, and *Performace Engineering* courses and books.

Programming Languages Taxonomy in CS

There are a lot of programming languages these days. One way to analyze programming languages is from the angle of the implemented type system (**type** are two sets: objects and operations with them. This concept is close to the concept of Algebras in mathematics; even in mathematics, Algebra operations are closed under the underlying set, and for **programming language types**, it's not necessarily the case):

- **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)
- Implicit conversions between types are not allowed (Strong Type System)

What is Object Orientated Programming (OOP) in CS

Object-oriented programming requires some special way to organize code, but it does not force it to have a `class` keyword in the Language. Please check this Appendix if you're curious about what OOP is from a Computer Science point of view C++ Technical Note/Object Orientated Design, which is enough for any practical purposes.

Philosophy of Python

If you want to understand some hidden built-in principles inside Python language, then after having a workable version of Python Interpreter, read the output from the following command, which Python developers have left as Paschal Egg:

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

This provides a point of view of the world from Python's point of view.

If Arriving at Python from C++

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++) programming language does not contain a native event-based system, even though there are frameworks on top of it that support that (for example Qt).

Meaning of Object

There is a clash of terminology if you have a C++, Java, or C# background.

In Python, everything that takes up memory in some form is called an **object**. In other languages (Java, C#, C++), an 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, but all entities that take some memory are objects in Python terminology.

One way to get access to all objects in the Python interpreter is via utilizing gc.get_objects() API in the following way:

```
import gc, sys

class ExampleObject:
    pass

obj1 = ExampleObject()
obj2 = ExampleObject()
obj3 = ExampleObject()

checedType = ExampleObject

print("Available object with type: " + str(checedType))
for obj in gc.get_objects():
    if isinstance(obj, checedType):
        print(obj, " | SIZE IN BYTES: ", sys.getsizeof(obj), "|ID:", id(obj),
"|TYPE: ", type(obj))
```

Subtleties with Accessing Value of Variables

Python has named references to objects. It has no variables available to the programmer for reference by value (at least at the Language level). Functions return and accept arguments by reference in the terminology of C++, C#, and Java. Consequently, you do not have *direct* access to the raw view of underlying value.

Moreover, there is still one not-nice aspect with the default argument presented in Python since the early `2.*` version. For the default argument, the default value 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, and the default value is just shorthand not to write arguments in some circumstances.

Context in Python as Scope 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.

In fact, in Python, there are only 4 contexts (which are, in C++ terminology, called scopes):

- **Local Context/Scope.** The context inside the current function. As we have mentioned earlier, a new loop with a new indentation does not introduce a new local scope as is happening in C++ with using curly braces `{}`.
- **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 are built into the interpreter.

Pointers and Object Reference Equality

As we have said, an Object in Python is everything that takes memory. Fundamentally, in Python, Object equality can test one of two things:

Value equality - what is tested is the fact that the objects have the same content. The programmer can define what this means by defining **eq** ina user-defined class. This operator semantically is the same as `==` in C++.

Identity equality - testing when two references are referenced to the same object. The programmer cannot determine what it means, and it is defined by the Language. Such type of equality is used during the use of operators is, is not, or you can explicitly request identity via the built-in function `id()`.

In Python 2/3, there are no pointers. However, the built-in function called `id(obj)` according to [1],[2], link means the address of an object in the interpreter's virtual memory.

> 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 guaranteed by the standard because there is no standard (and this is a cons. of not having the standard).

Therefore, in principle, the fact 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()`.

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, virtual keywords, and virtual tables in Python at all. In fact, in Python, methods are *attributes* that are bound dynamically at runtime, and function invocation happens dynamically according to all tutorials, at least before Python 3.11.

In very specific circumstances, you can even manually remove, get, and set attributes:

- delattr() - remove an attribute
- hasattr() - checks if an attribute exists.
- setattr() - set the value of an attribute
- getattr() - get the value of an attribute
- dir(obj) - collect the list of attributes for object `obj`.

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

> Payment for this flexibility - the class methods invoking mechanism implementation is suboptimal for Python runtime compared to available options for C and C++. Such flexible design embedded into scripting language is targetting for comfortable use and we understand that sometimes this flexibility to extend libraries is pretty nice to have, but this flexibility is not free. The price you pay is suboptimal time to invoke the function.

Language Differences Between Python and C++

1. Assignments opertor `=` in Python does not (semnatically) copy the actual data. The language just binds a right object to the left operand of the `=` operator. So the equal operator `=` does not copy data; it reassigns object reference.
2. In Python class, data attributes override method attributes using 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.
3. Essentially, nothing in Python makes it possible to enforce data hiding; it is all based on convention.
4. In Python, when using logical connectives, the result of a compound expression equals 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
```

5. Python has the `elif` keyword. For Python, it's crucial because syntax rules require creating indentation between `else` and `if`. The absence of such a keyword will potentially make the source code less readable. C++ contains parenthesis '{', '}', and such operators are absent in Python. In the experience of the authors of this note, explicit blocking in code makes it more readable (but there are technical debates around this aspect). Example:

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

6. In Python you may have the else branch inside the loops. If you did not read the Python Tutorial [1] (or it was a long time ago and you need a refresher), it can be the case that you never heard about it or forgot about it. This logic in the else path of a loop statement is executed when the loop terminates through exhaustion of the iterable or when the condition becomes false. However, the `else` path will not be executed if the loop has been terminated early with a `break` statement. See also: https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops.
7. In Python there is the "exotic" operator `**`. This operator can raise integers, real, and complex numbers to specific powers. Such a built-in operator is absent in C++.

8. Unlike many C-based languages, Python does not have a unary post(postfix) `++` increment or `--` decrement operators that increment or decrement integer operands by one. The reason for having this inside compile-based languages is because for such construction, there is typically just an Assembly command.
9. The boolean literals are named as `true`, and `false` in C++. However, in Python, they are named as `True`, or `False`. The underlying type `bool` has the same typing both in C++ and in Python.
10. In C and C++ you can implicitly cast expressions in conditions to bool or integer type. In Python the following expressions are considered `False`: `None`, `0`, `empty sequence`, `False`.
11. Python uses `.` to separate packages/subpackage/classes like C#, and Java, while C++ uses the namespace resolution operator `::`.
12. In Python (and in Perl), there is no such thing as function overloading, while C++ provides function overloading.

Sources of Confusion between C++ and Python Software Engineers

Python is a useful and nice *scripting language*, which is, of course, more comfortable to use compared to Perl (https://www.perl.org/books/beginning-perl/) or Windows Batch (https://en.wikibooks.org/wiki/Windows_Batch_Scripting) and sometimes more comfortable to use compared to Bash (https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html). In our experience, Bash is still easier to use if the script contains more commands for executing external processes for different tasks and less logic, but with proper wrapping of `retValue = subprocess.call(cmdline, shell=True)` it's possible to achieve the same equivalent.

Python has a very fast learning curve, which opened the doors (fortunately) for people not only in CS to create useful scripts in day-to-day life. The need to democratize program creation [software development] was realized by Peter Norvig a decade ago.

> 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 "Language" and what is "Programming" by itself.

Let's start with the "Language" concept in the context of Programming. One meaning is that Language expresses ideas with defined formalisms, and you *don't care at all* how these ideas are materialized in the real end system.
Even though it may sound reasonable, it's not the only definition. The message that one scientist from the programming languages area (B. Stroustroup author of C++) tries to bring for people for already 3 decades is that *Programming Language* is the language that gives you a way to describe the algorithm in a form that this algorithm will be executed in the computing device.

With this very strict definition Python, Java, C# are not Programming Languages. Python is a receipt for an interpreter, while C# and Java are runtimes coupled with a Just-In-Time(JIT) compiler. We do not want to offend any scripting language, but we want to highlight that there is a problem with the definition of the meaning of what is a programming language in the first place.

If you only start with Programming in your career or you are from another domain and not from CS exactly, or if you are from CS but with
no experience in Compilers/OS/Systems you may not see the difference. But there is a subtle fundamental difference.
It does not say that scripting and just-in-time compiled-based languages are incorrect, but at least please understand it's not the computer that executes this program, but what is executing is another (one more) level of abstraction.
The catch is that, in fact, any level of Abstraction is not free in terms of consumed memory and execution time (if this abstraction is used during runtime, not during design time).

> Python is a general-purpose programming language.

It is, at the same time, a very strong statement and, at the same time, a vague statement. There can be three points of view on such a statement:

1. By general-purpose programming language, you mean that you can create any Algorithm in it. If this is a definition, then it's correct. Only DSL languages constructed for special purposes in practice may have a lack of being Turing Complete because they were designed for only one specific purpose. To have one mechanism that is sufficient for expressing any computable functions is a mechanism that allows expression sequencing (executing basic blocks of instructions sequentially), selection (selecting between several blocks for execution based on condition), and iteration (transfer control to a previous statement or beginning of a basic block). For details, see Structured Program Theorem.
2. By general purpose, you mean that you can use it across many domains. In this case, how many are "many"? Depending on the definition of "many" Python may lie in this class and may not.
3. By General Purpose, you mean you can create programs for all programmable computing elements in the computer. If this is your definition and you believe that Python can help with this, then it's *wrong* (at least in our opinion). Python by design is the replacement for Bash, it's not a replacement for C and C++, or any other traditional compile language. Counter example: You cannot create drivers for your devices in Python.

> Python is more elegant and shorter than C++.

One more time, it depends on what you mean exactly. Some logic that works with byte view in the virtual memory of the process cannot even be expressed in Python in an effective way. But if you consider application classes that can be expressed in Python, then in general, this is true if you measure elegance by the number of lines of code in Python.

However, the first thing in creating algorithms is that they should be correct. In our experience, what is interesting after some amount of code is that there is a very strange asymmetry that you will observe once you create projects in Python and C++ with 40K lines of code and more. At least, such a big Python is not even close to C++. Compiling languages forces you to follow some discipline, and Python does not. If you have high professionalism in Python you may induce this discipline in some another way, but not at the language level. There are tools for Python that help (e.g., pylint), but a compiler and linker are far more powerful tools for detecting errors than any static analyzer applied for the language with the dynamic type system.

> Python is everywhere

This overstatement can also be read from the Python Tutorial. Please be aware that (any) interpretable language that exists or will be created in the future will have the following downsides:
C++ Technical Note / Downsides of Interpretable Languages. If you are creating your own scripting language, please be aware of this.

Python Basics

Language Benefits

It depends on your point of view and your style, but there is a point of view where the following things presented in Python are benefits. They shine especially if you have limited physical time to finish a project in a social sense:

1. More correct code from a style point of view.
2. Automatic cross-platform serialization or "pickling" for the user and built-in types from the Python standard library. https://docs.python.org/3/library/pickle.html
3. A lot of free and commercial IDE with IntelliSense:
    - PyCharm
    - Visual Studio Code
    - Visual Studio Python Extension
    - Komodo IDE
4. Python uses <`.`> token to separate packages/subpackages/classes like C# and Java. Packages from Perl are called modules in Python.
5. There exists an extremely big collection of extra modules for Python. The official pypi index storage has around 540K projects May, 2024)
6. Built-in Python pdb debugger.
7. There is no need to configure a compiler, linker, etc. (collectively known as a toolchain in Software Development).
8. There is no need to configure any build tool. The issue is that while using highly flexible tools like CMake or GNU Make to build your project using one of language supported by GCC you have to spend plenty of time to figure out how to work with these tools (if you do not have this background already).
9. Parsing of the function in Python is performed only at the moment of the **direct call**; there is no compilation phase.

> What you gain with (8) and (9) is the ability to start a project fast; what you lose is subtle control of how your source code of the program materializes into the executable program.

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. See also documentation: pdb: The Python Debugger.

How to Start Interpreter

Based on [1], the Python interpreter operates somewhat like the Unix shell. There are four ways to start the Python interpreter:

1. When it is called with standard input connected to a device, it reads and executes commands interactively.
2. When a Python interpreter is called with a file name argument, it reads and executes a script from that file. For details, see Python Tutorial / Interpreter
3. You can start the interpreter by calling

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

   It executes the statement(s) in the command, analogous to the shell `-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.
4. Some Python modules can be used 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.

Script Arguments. The script name and additional arguments are turned into a list of strings and assigned to the `sys.argv` variable with the following rules:

- 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.

What is a False Statement

The following expressions are considered `False` in Python:

- `None`
- `0`
- Empty Sequence: An empty sequence is a sequence for which `len() == 0`
- `False` logical expression: Logical expression which gives `False` as a result

First Line in Your Script

The line `#!/usr/bin/env python3` in a well-developed script (for POSIX OS Family) contains what is known as sha-bang. It has a long history in Unix/Linux OS, and it is used to understand which interpreter to use from an OS perspective. For Windows OS It's possible to use it on Windows OS as well. In Windows, a binary application named `py.exe` is a launcher that performs a choice of a used interpreter based on the mentioned sha-bang.

Formally, this line is optional.

Possible Second Line and Source File Encoding.

The source file by itself is represented in characters. Characters that constitute the file content can be in any supported encoding. By default, Python source files are 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:

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

Physical and Logical Lines of your Script

Physical lines in the source code of a script are physical lines inside the text file encoded with one of the possible encodings.

A **Logical Line** of a script 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.
- Normally you should use `\` for line continuation. However, Python automatically supports multi-line continuation inside the following constructions, and when you are inside such an expression, you can split your expression over more than one physical line without using the backslash symbol:
    - Expressions in parentheses `(arg0, arg1)`, e.g. which is used for passing arguments to function invocation or when you define a tuple.
    - Expressions in square brackets `[item0, item1]`, e.g., which is used to initialize the built-in list object.
    - Expressions in curly braces `{setItem0, setItem1}`, e.g. which is used to define built-in sets and dictionaries.

A parser reads a Python script. The parsing of some pieces of Python source code (e.g., of the function), as we already mentioned, is performed only at the moment of the direct call of this function. I.e., if you have errors in the function, but you did not invoke it, you will not observe problems a-priori (from the Python interpreter side).

Comments

Comments in Python start with the hash character `#` and extend to the end of the physical line. One workaround for the limitation of supporting multiline strings is to create an unnamed string inside the function body, but not as a first string; see this tweet by Guido Van Rossum: https://x.com/gvanrossum/status/112670605505077248 from 2011.

Operator Precedence

Based on C++ Technical Note/Lexical Analysis - in the terminology of Programming Languages, tokens are separate words of a program text. In most programming languages, the tokens can fundamentally be one of the following types:

- a. Operators
- b. Separators
- c. Identifiers
- d. Keywords
- e. Literal constants

So, operators in almost all Programming are one of the five fundamental language concepts.

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

Simple Built-In Types

- **Ellipsis.** There is a single object with this value. This object is accessed through the built-in literal `...`. If the expression `...` is used inside the condition, then `...` is implicitly converted into `True`.
- **NoneType.** This type has a single object. 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 this expression is used inside the condition, then `None` is implicitly converted into `False`.
- **Integers (int)**. The type that is used to represent integers in the Python interpreter does not have any fixed number of bits. Instead, it has a varying size. In this sense, an integer has an unlimited range. You can practically hold as big an integer as you want until you start having problems with virtual memory in the interpreter process.
- **Real (float).** This type represents 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 for a language. Of course, for people involved in scientific numerical fields, such a statement may be incorrect in some circumstances.

Documentation on this subject: https://docs.python.org/3.13/reference/datamodel.html

Simple Statements

Typically, a simple statement is comprised of a single logical line. However, several simple statements may occur on
a single logical line separated by semicolons. Example:

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

For a more precise definition, see: https://docs.python.org/2/reference/simple_stmts.html

Compound Statements

Generally, compound statements are written on multiple logical lines using indentation. However, 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 (3); #legal python code
```

which is equivalent to:

```
if 1:
    print (2)
    print (3)
```

Empty(Pass) Statements

The `pass` statement does "nothing" similar to the C++ `;` statement. Such empty statements are used in programming languages when the language requires a statement, but the logic of the algorithm requires "do nothing".

Division of Numbers in Python

The division operator in Python 3.* `/` 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.

Printing

The standard way to print something in Python 3 is by utilizing the built-in print() function.

There exists 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 a string interpolation feature has been added (https://www.python.org/dev/peps/pep-0498/) into the Python scripting language.
It is presented in Bash and similar scripting languages, however syntax may vary from interpreter to interpreter.

Example:

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

The f-string form is known as a formatted string literal. To use formatted string literals, you should begin a string with `f` or `F`, before the opening quotation mark in the form of <`'`>, <`"`>, <`"""`>, <`'''`>.

Inside this string, you can write an arbitrary Python expression between `{` and `}`. Also, you can refer to variables or literal values. While using the f-string, you can pass an integer after the `:`. It will cause that field during printing to have 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 that can be useful during debugging. The `=` specifier can be used to expand an expression, too:

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

Example:

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

Output:

```
a=123
```

String interpolation (named in Python as **"formatted string literal"**) is evaluated during script execution. In terms of type, the f-strings are just usual string `str` and it's not a new type.

Finally, old string formatting from Python 2 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` that returns the successive items of the desired sequence when you iterate over it. It does not make the list, thus saving space. In the old days, the Python language created a list with `range()` and implicit sequence with `xrange()`, but it was a lot of time ago in Python `2.*`, and currently we can forget that such a situation had a place to be.

When you are looping through dictionaries, you are enumerating through `keys`. If the key and corresponding value are important to you, then you can retrieve the value and key at the same time. To do 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 order, 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 About Conditions

The conditions used in `while` and `if` statements can contain any operators but typically contain:

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

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 `||`. The boolean negotiation carried with the operator `not` is equivalent to the operator `!`.

> Chaining questions can be used to "kill" candidates in technical screening. Example:
>
> ```
> mary_songs = [576093, 297863, 871532]
> index = 297863
> print(index == index in mary_songs)
> ```
>
> If you don't know about the chaining rule you may expect that results are `False`, but it's actually `True`.

Next, Python supports the same set of the bitwise operators as it does in C/C++ languages (See operator precedence), and it includes:

- `&` - for bitwise and.
- `|` - for bitwise or.
- `<<` - left bitwise/arithmetic shift.

- `>>` - right arithmetic shift with sign extension.
- `~` - operator yields the bitwise inversion.
- `^` - for bitwise xor.

> Side note: in C++ `>>` from C++20, the right shift on signed integral types is an arithmetic right shift. However, before C++20, it was implementation dependent.

In Python, when you use an expression (not necessarily Boolean) that internally uses Boolean operators, 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):

```
#/usr/bin/env python3
# Python

a=123 or 12
print(a)
# Output: 123
```

```
// C++

#include <iostream>
int main() {
  int a = 123 || 12;
  // Alternative notation:
  //   "123 or 12" is rarely used in C++
  //   https://en.cppreference.com/w/cpp/language/operator_alternative

  std::cout << a;
  // Output: 1
  return 0;
}
```

Basic Data Types

Python has several basic data types including integers, floats, booleans, and strings. These data types behave in ways that are familiar to other programming languages. As it has been mentioned in Python 3.11/Glossary an object's type is accessible as its **class** attribute or can be retrieved with `type(obj)`.

```
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"
```

Boolean Variables and Operators

Python implements all of the usual operators for Boolean logic, but it uses English words rather than special symbols (such as `&&`, `||`, 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 also 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"
```

Built-in Containers

Python includes several built-in container types:

- Lists
- Dictionaries
- Sets
- Tuples

Containers are devoted to storing values. A list is the Python equivalent of an array (conceptually, even inside the Python interpreter, it's implemented as a list that can be checked from the source code of the interpreter). Python lists are (i) resizeable and (ii) can contain elements of different types. Examples:

```
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 the sublist.

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]"
```

Deletion from the list and other containers happens via using del statement in the language, which invokes `__del__`
special method in the class definition (link).

Comparison of Containers

Sequence objects (typically) can be compared to other objects with the same sequence type. The comparison uses lexicographical ordering to define order relations. If, during comparison, the type of the objects does not support the comparison operator, the exception is raised.

Strings

Python has great support for strings:

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # And string literals can be enclosed in double quotes
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"
```

Python can manipulate strings, which can be expressed in several ways:

- String enclosed in single quotes (`'`...`'`) can use double quotes <`"`> inside the string literal.
- String enclosed in double quotes (`"`...`"`) can use single quotes <`'`> inside the string literal.
- String enclosed in triple quotes (`"""`...`"""`) or (`'''`...`'''`) is a multiline string. Multiline strings can be placed in several strings. In this case, the used new line character is part of the string literal.
- The raw string literal is represented as `r"`...`"` or `r'`...`'` or `r'''`...`'''` or `r"""`...`"""`. Inside the raw string, you can use backslash characters in the usual way. In this mode, the backslash inside string literals does not have any specific meaning. The raw string notion is similar to C++11 construction `R"(hello\n)"`.
- Two or more string literals next to each other are automatically concatenated without using the plus sign. This syntax and semantics coincide exactly with C++ and C.

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 the 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))
```

Python Technical Details: One Step after Basics

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 other words, Python does not have interfaces as a *pure language concept*.

Instead of a specific and robust (but more long and verbose) interface notion, the Python scripting language (and other scripting languages as well) 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 the object does not support a specific interface.
In this latter case, the AttributeError will be raised.

If you define your classes in Python and you want the objects of your class to be used in specific language construction like iteration, you should support **protocol**.

Next, we will go through the main protocols:

- **Container protocol.** Built-in container types in Python
  tuple, list, str, dict, set, range, and bytes all support `in` and `not in` operations. The support of `in` and `not in` is known as 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)
  ```

- **Iterrable protocol.** Built-in container types in Python supports the calling of iter(x), next(x) operators. Such objects can be used in for-loops:

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

  For your types to provide the ability to use these built-in functions, 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 classes, you should define __getitem__, __setitem__, __delitem__.

  Once you define these operators, you can use the following language operators:

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

Introspection of System

One way to get information about the interpreter version:

```
python --version
```

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

```
import os, platform, socket
print("=====================================================")
print("Information about your system")
print("=====================================================")
print(f"Python interpreter: {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())}")
```

Introspection of Python Objects

As we said in Python terminology, everything that takes some memory 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 current object or module if apply dir for module link.
- `__doc__` - documentation string for the object link.
- `__class__` - type name of the object link.
- `__file__` - the name of the source file here module has been defined. link
- `__dic__` is a dictionary used to store an object's attributes. Not all instances have it, but typically, it's the case. link

Example:

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

Comprehensions Syntax

Comprehensions Syntax provides a short syntax to create several iterable expressions with short syntax. List comprehensions in Python:

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

Set comprehensions in Python:

```
{expr(item) for item in iterable_container if condition (item)}
```

Dictionary comprehensions in Python:

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

Generator comprehensions in Python (more about generators is generators):

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

List comprehension is described in the Python Tutorial [1]:

- List Comprehensions
- Nested List Comprehensions

List and Set Comprehensions

When programming, we frequently 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]
```

As we have already implicitly described the 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)
```

Also, in Python, there is a syntax for Dictionary comprehension. 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 (in Python) 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.

Tuples are created using parenthesis in the following form: `(elementA, elementB)`.

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"
```

Working with Tuples

A tuple consists of several values separated by commas. It is not possible to assign to the individual values of tuple elements, 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 a special syntax rule. We should construct it via a "value" followed by a comma (it is not sufficient to enclose a single value in parentheses). And so, creating 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:

```
b = tuple([1,2,23])
```

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

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

Unpacking of Containers

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 `**` to unpack key-value pairs.

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

Use the `zip` function to construct a dictionary from two separate lists with keys and values:

```
keys = ['a', 'b', 'c']
values = [1, 2, 3]
print(type(zip(keys, values)))
# Prints: zip

d = dict(zip(keys, values))
print(d)
# Prints: {'a': 1, 'b': 2, 'c': 3}
```

Functions: Introduction

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']!"
```

This is an introduction to the function definition.

We will see more details about functions in details a bit later.

Random Interesting Constructions

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 a 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 that is built-in in the Language is set. On sets, you can perform set-theoretic operations. Example:

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

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

```
emptySet = set()
```

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

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

   Documentation: https://docs.python.org/3/reference/expressions.html#conditional-expressions

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

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

8. Multiple assignments in Python:

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

9. 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
```
```

10. A deleted statement is a statement with `del` built-in operator. It can be used in two different contexts:
    - Delete statement `del` is used to delete elements from a container such as a list.
    - Also, it is used to delete variables from the interpreter. After variable deletion, the variable is not defined. If a variable is not "defined" it means it has not been assigned a value or has been deleted. Trying to use it for reading will give you an error. Trying to use it for writing 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.

Classes

User-Defined Classes in Python in C++ terminology

Firstly, Python has limited support for private attribute objects.

When you name attributes as `__attributename`, the interpreter performs name mangling. In reality, in case of accessing this attribute outside of the class, it will have the name `_classname__attribute`.

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

1. Normal class member,s 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 user-defined classes
4. Python Data attributes correspond to *"data members"* in C++
5. Python supports multiple inheritance
6. Python supports exceptions
7. In Python (and in Perl), there is no such term as function overloading

The Syntax for Defining Classes

The syntax for defining classes in Python is straightforward and has the following 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 "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"
```

Technical Details about Language Concepts

Convention about Variable Names

Convention	Example	Meaning
Single Leading Underscore	`_var`	This naming convention indicates a name is meant for internal use. Generally (with one exception) not enforced by the Python interpreter. In this case, it was meant as a hint to the programmer only. The exception case is the behavior of wildcard imports. 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 for variables. The Python interpreter will automatically use correct mangling inside methods of the class. Outside the class, Python interpreters will not perform any name mangling.
Double Leading and Trailing Underscore	`__var__`	Indicates special methods defined by the Python language. Avoid this naming scheme for your attributes if you use it just for variables.
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).

Variables Introspection

There are several ways to take a look at available symbols:

- globals() - will give you a dictionary of current available 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, then this function will not show you globals.

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']
```

These variations exist in the first place because there are several different contexts/scopes in Python as has been described in Context in Python Section.

Global and Nonlocal Variables

In the previous example from the Variables Introspection section we have used nonlocal, and global keywords. Now, let's take a close look at its meaning.

global.

Documentation: https://docs.python.org/3/reference/simple_stmts.html#global

`global_stmt ::=  "global" identifier ("," identifier)*`

The global statement is a declaration that holds for the entire current code block scope.
This declaration means that the listed identifiers will be interpreted as global variables. Global variable scope changed immediately to the module-level binding without any intermediate outer scope, relative to the current scope. It is impossible to assign value to a global variable without using `global`. However, you can actually (formally) refer to global variables if you want to only *read* from them.

nonlocal.

Documentation: https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement

`nonlocal_stmt ::=  "nonlocal" identifier ("," identifier)*`

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

You should use these variable access modifiers when you're going to *write* to the variable and provide an interpreter a hint about what you are going to do exactly:

- You want to define a new local variable,e and you provide the default value with the `=`.
- 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.

In C++/C#/Java, such a *nonlocal* scope is impossible because, in these languages, you cannot define a function inside another function.

At the syntax level in C++, you actually can define a lambda function inside a function, but it is a syntax sugar.

You can read more about Lambda C++ functions from [C++ Technical Note from C++1998 to C++2020/ Lambda Functions](). In C++, you cannot define a function inside a function because by design of the language, it does provide extra expressibility. Similar to the idea of a "universal base class," the concept of having nested relations between functions does not make any sense in the C++ mentality.

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`. Python module files are files written with Python language, and in general, they look like usual scripts. Within a module, the module name (as a string) is available as the value of the global variable `__name__`.

Typically, you can launch the Python module as a script if the authors of the module provides some utility functionality either with `python script_name.py` or `python -m script_name`.

During launching the module, the code in the module will be executed, just as if you imported it, but with one difference the **name** will be set to "**main**".

The part of the code for executing the script functionality of the module typically has the following guard condition:

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

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

To import the module, the following instructions should be (typically) used:

```
import my_module
```

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

Next, we would like to say that each module has its private namespace, which is used as the global namespace by all functions defined in the module.

In the form of an import statement in the form of:

```
import my_module
```

In this form, all global symbols are defined in `my_module` and start to be available through

`my_module.<function|variable name>`.

The import statement does not add the names of the functions defined in `my_module` directly to the current namespace. However, there is a variant of the import statement that imports names from a module directly into the importing module's namespace:

```
from my_module import func_f, func_g
```

In this form of import statement, the imported module name itself is not introduced into the global namespace of the current module. The next variant is to import all names that a module defines:

```
from my_module import *
```

It imports all symbols except variables and functions whose names start with an underscore `_` from the imported module and are imported into the current module global namespace.

A module typically contains function and class 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.

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 can be used to find names that the module defines.

When you launch any Python scripts, if all is OK in the Operating System(OS) and you have enough privileges inside the OS, then the script will be launched and executed. However, it is important to mention that the `import` statement (by default) is executed only once per Python session. If you want to reload the module (e.g., because the source code has been changed and you don't want to relaunch the application), please consult the Module Reloading section.

Packages

Packages are a way of structuring Python's module namespace by using "dotted module names" in a hierarchical way.

Firstly, you should have several source code files with modules. Next, to group these modules into the package, you should place modules in one directory and create the file `__init__.py` in this directory. The `__init__.py` files are required to make Python treat directories containing the module source files 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.

Let's execute the code:

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

Specifically in this case, the `__all__` variable defines the modules that you should import to a client who has requested this [import wildcard form].

Reference between Modules in Packages

There is a specific way to use relative addressing schemas between modules via an intra-package reference schema. The intra-package reference 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 in the filesystem folders hierarchy:

```
from .. import formats
```

Rules for Search Modules and Packages

When you import a module via keyword [import], the runtime importing the module via the following rules:

1. The interpreter first searches for a built-in module. The built-in module names are built into the language and are listed in `sys.builtin_module_names`.
2. Next, the interpreter tries to find the module in a list of directories given by the variable `sys.path`. * The variable `sys.path` after starting the interpreter contains:
    - The directory containing the input script (or the current directory)
    - Paths in environment variable `PYTHONPATH`
    - The installation-dependent default paths, such as the site-packages directory
    - Python scripts can modify the `sys.path` variable itself during runtime.

About Functions: Now in Details

About Indentation

As you know, Python was designed using indentation, and there are no scopes constructed with `{}` as in C-based languages (e.g. C, C++, etc.). During writing language construction, sometimes, the indentation style can be shorter. But sometimes, when you have 3-9 nested loops of the algorithm logic, the indentation makes the logic less readable and more error-prone.

In principle, you can use both spaces and tabs for indentation, 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

Function Body

The first statement of the function body can optionally be a string literal - this string literal is the function's 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 values 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.

Function Arguments and Return Value

The actual parameters (arguments) to a function call are introduced by themselves 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 a value from a function. Return without an expression argument returns `None` implicitly. Falling off the end of a function also returns `None` implicitly.

Default Argument Value

You can specify a default value for one or more arguments.

> Warning: The default value is evaluated only once, and by itself, the default value is a *"variable"*.

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 (usually) 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 Arguments

```
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` or `tuple` object with `*` for positional arguments only
- Expand the dictionary dict object with `**` for keyword arguments only

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}
```

Lambda Function

The 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. In contrast to C++, there is no need to write a return statement:

```
#!/usr/bin/env python
sum = lambda x,y: x+y
```

Which is the same as the following lambda construction since C++11:

```
auto sum = [](int x, int y){return x + y;};
```

And the following C++14 generic lambda construction since C++14:

```
auto sum = [](auto x, auto y){return x + y;};
```

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 a Python expression evaluating the value of the annotation. Return function 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 metadata in its logic).

General Convention uses 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 as a software pattern decorate target function execution. A decorator is a function that returns a value 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 decorator:
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.

Common examples of standard decorators that can be observed in Python codes are:

- @staticmethod - which is a class method that **does not** receive an implicit first argument as a reference to the class object.
- @classmethod - which is a class method that **does** receive an implicit first argument as a reference to the class object. Therefore, the "class method" has as an implicit first argument, which is a class type.
- @property - this is a decorator for properties. (see Appendix Object Orientated Design Patterns in Technical Note From C++98 to C++2x.) that turns the method into a "getter" for a read-only attribute with the same name.

Classes in Python

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

The method function is declared with an **explicit** first argument representing the object, provided implicitly by the method call and by convention, typically called `self`. For comparison, traditionally in C++, the reference to the object is not declared by the programmer, and the pointer to the object is accessible through `this`. Only recently, in C++23, the alternative syntax with explicitly declaring the `this` object has been added to the language. See Technical Note. From C++98 to C++2x/C++23 - Language Features.

Like in C++, most built-in operators with special syntax can be redefined for class instances.

Unlike C++, built-in types in Python can be used as base classes for extension by the user.

To inspect if the object is an instance of some class, you can use isinstance(obj, classinfo), which is the analog of C++ dynamic_cast.

To inspect class relationships, you can use issubclass(classDerived, classBase). This functionality is absent by design in C++. The C++ style says if you need it, you should implement it and pay for it with computing time and memory.

```
class A(int): pass
print(issubclass(A, int))
# Print: True
print(isinstance(int(), int))
# Print: True
print(isinstance(A(), int))
# Print: True
```

For another built-in function, see: https://docs.python.org/3/library/functions.html

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
      DerivedA.f(self)            # Call DerivedB explicitly

obj = DerivedAB()
obj.f()
```

Magic Methods for Classes

> Magic is an official term used by the Python community, even though in professional and science literature this term is used rarely.
>
> This informal name shines a light on the fact that a lot of things inside the Python community happen informally without standardization.
>
> By the way, it does mean that this is bad and wrong...
>
> Python is not the only place where such a situation exists. The effect that both styles (formal and informal) can coexist can be obtained by looking into API and development styles for Android OS and Linux/Windows OS or IOS. It is well known that the development for Android OS is mostly cowboy style.

A class can implement certain operations that can be 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 module source code, you have two options on how to reload this module:

1. Restart the Interpreter.
2. Use reload() function from imp module which provides access the import internals. Example:

```
import imp
imp.reload(module_name)
```

Encoding During Reading Files and With Statement

The text file is iteratable, 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,g you can specify this in open().

To open and close files, you can use open()/close() calls. One shorthand to automatize this operation is to use with statement.

The `with` statement provides two facilities:

- The opening of the file.
- Closing in the successful execution of its block and closing in case of obtaining any software exception within the `with` block statement.

Example:

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

The reason for closing files and closing in general is the following. Some objects contain references to external resources such as open files in the filesystem. 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. This design demonstrates that Garbage Collection is not a universal solution for all situations and various types of resources available inside the process/script runtime.

Defaultdict

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

Defaultdict is working mostly like a std::map in C++.

Specifically, 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 for a type specified for defaultdict.

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 (which has been included in Python 3.10 language https://docs.python.org/3/whatsnew/3.10.html) 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 `_` in `case` acts as a wildcard and never fails to match, similar to `default` in switch statement in C and C++.

Once a match is found and the corresponding branch of code is executed, the match expression ends.

In other languages, it is achieved using the `break` keyword, which is not needed in Python. In other words, fallthrough behavior is not presented in Python in the first place. Therefore, for people with C++, C, Java, C# backgrounds, it may add confusion.

Walrus

In Python (unlike in C and C++), assignment inside expressions, which is used in the conditional statement, must be done explicitly with the walrus operator `:=`. It's the same as the operator `=` in C++ applied in the context of expression inside the `if` statement.

The operator `:=` in Python can be used *only* in the context when you want to perform an assignment inside conditions in the `if` statement. If in Python you do not use the walrus operator `:=`, but the usual assignment operator `=`, then it will lead to a syntax error.

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

# Output
# 1
```

This syntax has been included in Python 3.8 (https://docs.python.org/3/whatsnew/3.8.html).

Generators

The class-based iterators are a functionality that is implemented via two functions:

- `__iter__` method
- `__next__` method

See also Interface and Protocols in this document.

Generators in Python are a specific type of function that uses the (`yield`) statement. What makes generators compact that `__iter__()`, `__next__()` methods are created automatically.

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 separate 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 does 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 yield statement
print(next(a))   # run to second yield statement, etc.

# The next yield will throw a StopIteration exception
```

Standard Tools and Some Libraries for Computing and Visualization

Package Managers

To have the ability to launch a big enough project in Python, you typically need to install extra libraries. The culture of Python script development (and, in reality, Python interpreter development itself) relies on the huge utilization of external libraries.

Two pretty standard ways to install libraries for Python are installation via `pip` and `conda` package managers.

The `pip` package manager is preinstalled with a Python interpreter. You may prefer not to use pip directly, but instead use the command for calling pip directly from Python:

```
python -m pip list
```

It can be useful to eliminate problems with various installations of Python interpreters in the system (if you have several interpreters).

An alternative to pip is conda. You can install conda in two ways:

1. Use Anaconda distribution containing Conda.
2. Install Miniconda (https://docs.conda.io/en/latest/miniconda.html).

If you want to select (2) way, then for example, you can use the following command if you are using Linux distribution for x86-64 compute architecture:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b
export PATH="${PATH}:~/miniconda3/bin"
~/miniconda3/bin/conda init bash && source ~/.bashrc && conda config --set
auto_activate_base false
```

This code snippet has been taken from this OpenSource project: https://github.com/burlachenkok/flpytorch/tree/main.

It turns out that `conda` has two different goals:

1. Package Manager.

2. Environment Manager.

In fact `pip` by itself has only package manager functionality. Next, let's take a look into a package manager commands comparison of these two tools:

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

|

Links to Relative Information:

- 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/
- Standard packages repository named as "Python Package Index repository": https://pypi.python.org/pypi

Environment Managers

Each project may have a specific requirement in a specific version of the library, and this is the motivation behind the environment manager idea, which is presented in the Python ecosystem.

The `venv` module is a standard virtual environment in Python. However, another environment manager that is extremely popular is `conda`. 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 installed in the current environment
7	conda env create --file myenv.yml	pip install -r requirements.txt	Install all packages from requirement list or exported environment
8	conda remove --name myenv --all	Remove directory in filesystem with environment and it's scripts	Remove environment completely

|

Python Notebooks: General

Jupyter Notebooks present 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 the means to debug Python Notebooks; however, some IDEs such as Visual Studio Code support the debugging of code inside Python Notebooks.

Documentation about Jupyter is available here: https://docs.jupyter.org/en/latest/.

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

```
python -m notebook
```

Start the web server and allow incoming connections from all network interfaces to be accepted:

```
python -m notebook --ip 0.0.0.0 --port 9999
```

In recent versions, to skip the password requirement, you may want to use the:

```
python -m notebook --ip 0.0.0.0 --port 9999 --NotebookApp.token='' --
NotebookApp.password=''
```

> If you have problems with accessing the web server, the first thing is to take a look at whether the port that you have specified to listen accepts 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 for connecting to it from outside. 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
> ```
>
> It's bad for production but fine for development or prototyping.
> Alternatively, you can create an SSH tunnel from your local machines (from which you create the SSH session) to a remote machine that listens to a specific port in Jupyter Notebook (for example 2222) via the following command: `ssh -L 0.0.0.0:2222:remoteHost:2222 yourUserName@remoteHost`

The Python notebook files' extension is `*.ipynb`. The notebook contains code snippets and Markdown text organized by cells. Therefore, cells can be one of two types:

- Code
- Markdown

Python Notebooks: 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 the next cell

Command	Description
\%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	Alternative 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 about another magic command for Jupyter notebooks: https://ipython.org/ipython-doc/dev/interactive/magics.html

PyTorch Resources

PyTorch is a big numerical package that is most often used for training (Deep Learning) Machine Learning models. However, it can be used in other situations. For example:

- You have computations in NumPy, and you want to port them to GPU
- You work in the domain when you have to deal with explicit mathematical functions. The function is pretty complex to compute partial derivatives explicitly, and you want to automatize this process via computing partial derivatives numerically, but in an effective way for reasonably big dimensions.

In both cases, PyTorch will help you.

Next, we will provide references to Torch documentation:

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 require grad modification	https://pytorch.org/docs/stable/tensors.html#torch.Tensor.requires_grad_

Description	Link
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
Module - Base class for all neural network Module/Layer	https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module
The module that 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
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 real values [from all possible range of real numbers] that are fed into softmax or similar layer in which the image(or output) is a probabilistic simplex.

Matplotlib

Matplotlib is a plotting library. This section gives 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 a little 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 the 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( )
```

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 at: http://docs.scipy.org/doc/numpy/reference/routines.math.html.

Numpy provides various functions for manipulating arrays; you can see the full list at: 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 an incorrect thing to be designed in the first place, you are not alone).

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

- Install it with your package manager, e.g,. 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 the array - is number of dimensions.
- Tensor (in Machine Learning / Deep Learning) - a name used to describe multidimensional arrays.
- Shape - description of dimensions for a multidimensional array organized as a tuple. Each dimension of a multi-dimensional array is called a "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 in [0,1)
sampled uniformly at random.
print(e)                     # Might print "[[ 0.42  0.52]
                             #              [ 0.78  0.12]]"

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 arrays 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 ranks than the original array:

```
# Create the following rank 2 arrays with shapes (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 ranks,
# 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 ellipses are used in NumPy to slice higher-dimensional data structures. It's designed to mean at this point, insert as many full slices (`:`) as necessary 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 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 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 data types 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 the Syntax for Matrix Multiplication

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

In NumPy, instead of using `*`, 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)
```

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 the matrix into different forms, you can use the 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 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 when you broadcast unintentionally. 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 a 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 a size greater than 1, the first array behaves as if it were copied along that dimension

Profiling And Compute Optimization

Collecting Preliminary Information About the System

To obtain a version of the Python interpreter, use the following command:

```
python --version
```

To collect a summary about your system running on **Windows OS Family**, please use the following command:

```
systeminfo
```

To collect a summary about your system running on **Linux OS Family**, please use the following commands:

```
#!/usr/bin/env bash

echo "Computer architecture:" $(arch)
echo "Python version:" $(python --version)
echo "Linux verion:" $(uname -a)
echo "Linux distribution specific information"
lsb_release --all
echo "Information about CPU"
echo "Number of processors in system:" $(nproc)
lscpu
lscpu --extended
echo "Physical and Swap memory"
free
echo "Current Working Directory: $(pwd)"
echo "Disk Partition size for Current Working Directory"
df -h .
```

Select Libraries Based on Benchmarks and Preferences

Python was originally designed as an extension to Bash, but nowadays (as of August 2023), the Python community is exploring new applications for this language. However, Python has some serious fundamental limitations, and some tasks might be impossible or inefficient to implement in this language. Why it is so fundamentally you can check this section [C++ from 1998 to 2020: Downsides of Interpretable Languages].

This situation with Python is mitigated by using *libraries/middleware/framework*. A decade ago, these terms in Computer Science had specific meanings, but nowadays, they tend to be used in a dirty sense. This implies that you should not differentiate between these terms.

If you are working in a specific domain and you use Python, then it can be the case that you will need to use external libraries. You might not be interested in the debate about which language to use for which situation, especially if your language choice is fixed (i.e., Python).

> The commitment to the library and the absence of the ability to write effective algorithms (in which you gain expertise) have serious implications, but it is out of the scope of this note.

Even in the case of deciding to use a third-party library, you will need to commit to a choice of the library.

There are social aspects that are sometimes even more important than technical:

- The size of the community
- Quality of documentation
- Number of bugs or rate of fixing them
- Existence of courses and books if the framework is huge
- Form of the learning curve

Of course, there are technical aspects:

- Quality of code snippets and examples
- Interoperability with other libraries. I.e., how easy and effective it is for two libraries to be combined. (E.g.: PyQt can integrated plots in Matplotlib).
- Can customize the framework with your form data and/or presentation formats

There are even deeper aspects of science and engineering:

- Support your target hardware effectively
- Support state-of-the-art algorithms in the field
- Support parallelization across nodes and utilize communication patterns effectively
- Support of streaming, i.e. make all work in one sequential pass
- Amount of Consumed memory
- Scalability for extra communication or computation hardware
- Last, but not least: *wall clock time*.

If the library represents a bottleneck, you may want to compare different solutions. One way is to read engineering blogs or academic papers that provide benchmarks for similar settings. The most straightforward metric is *wall clock time* for a sequence of experiments carried in the same flavor, but as we said, sometimes it is not *the most important metric*.

Be careful both in the selection and in the criticism of people who are motivated by not wall clock time.

For instance, for popular web frameworks for creating web services and websites, you can check the following benchmarks:
https://www.techempower.com/benchmarks/.

> Do not be upset if the Python solution is in the low list of benchmarks, such as those presented above. Python has another benefit.
> But if you want to *fight* in an honest fight with another technology/language, be ready for a (hard) *fight*, especially with well-prepared scientists and engineers.

Please research what solutions suit your needs and use these benchmarks as a first approximation. Keep in mind that benchmarks are not perfect, and they might not reflect your specific setting or problem size in your hands, but they can provide you with the first steps in selection.

Usage of Matrix-Matrix Multiplication

One general recommendation (which is mentioned in any applied Machine Learning courses these days) is the following. If you can cast your problem into operations from linear algebra supported by NumPy, then use vectors and matrices from NumPy instead of the explicit loop in the Python interpreter.

Speedup improvement from using matrices directly can be on the order of `10-20`.

Concrete Example:

```
import numpy as np
import time, os, platform, sys

def dumpProgramInfo():
    print(f"Executed program: {__file__}")
    (system, node, release, version, machine, processor) = platform.uname()
    print(f"OS name: {system}/{release}/{version}")
    print(f"Python interpreter: {sys.executable}")
    print(f"Python version: {sys.version}")
    print(f"Platform name: {sys.platform}")
    print(f"NumPy version: {np.__version__}")
    print()

def testMatrixVectorMultNumpy(N=5000, D=3000, C=50):
    # Matrix CxD
    W = np.random.rand(C,D)
    # List of vectors [D,1]
    vecList = [np.random.rand(D,1) for i in range(N)]
     # List of vectors [D,1] organized as "N" columns
    vecListInMatrix = np.random.rand(D,N)

    print("***********************************************")
    print("Test function name: ", testMatrixVectorMultNumpy.__name__)
    print("Matrix W Shape: ", W.shape)
    print("Vector List Length: ", len(vecList))
    print("Vectors Plugged into matrix. Columns: ", vecListInMatrix.shape[1])

    s1 = time.time()
    R1 = W@vecListInMatrix
    e1 = time.time()

    s2 = time.time()
    R2 = [W@(vecList[i]) for i in range(N)]
```

```
    e2 = time.time()

    print(f" Matrix-vector operations using matrices: {e1-s1:.5f} seconds")
    print(f" Matrix-vector operations using lists: {e2-s2:.5f} seconds")
    print("RESULTS")
    print(f" Speedup from using matrix-matrix mult. VS sequence of matrix-vector
mult.: {(e2-s2)/(e1-s1):.5f} seconds")
    print("************************************************")

if __name__ == "__main__":
    dumpProgramInfo()
    testMatrixVectorMultNumpy()
```

Output in my Machine with
`Intel64 Family 6 Model 165 Stepping 2 GenuineIntel 2304 Mhz` CPU:

```
Executed program: using_numpy_aka_vectorization_for_python_users.py
OS name: Windows/10/10.0.19041
Python interpreter: C:\Program Files\python3.9\python.exe
Python version: 3.9.4 (tags/v3.9.4:1f2e308, Apr  6 2021, 13:40:21) [MSC v.1928 64
bit (AMD64)]
Platform name: win32
NumPy version: 1.21.4

************************************************
Test function name:  testMatrixVectorMultNumpy
Matrix W Shape:  (50, 3000)
Vector List Length:  5000
Vectors plugged into the matrix. Columns:  5000
 Matrix-vector operations using matrices: 0.03790 seconds
 Matrix-vector operations using lists: 0.49867 seconds
RESULTS
 Speedup from using matrix-matrix mult. VS sequence of matrix-vector mult.:
13.15767 seconds
************************************************
```

> If your functionality is not expressible at all with affine operations (matrix multiplications and vectors scaling and addition) then you are out of luck with this technique of compute optimization for Python Language.

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 from transferring Python logic into Cython are between **2x** to **1000x**.

And it depends on how much you use the Python interpreter compared to situations where your code is in C or C++ Libraries, which Python uses.

> By the way, the notion of low-level and high-level languages is pretty vague. For people operating in the level of scripting languages, the level at which you create algorithms in C or C++ is considered informally as low-level. And for compilers writers C and C++ are already high level. Therefore, the notion of high (or low) level highly depends on context.

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 types of function definitions inside them:

- `def functions` - It is usual Python functions, and they are defined using the usual (in the context of Python) `def` statement, as in Python. **Restriction:** Only Python functions can be called from this function.
- `cdef functions` - C-like functions are defined using the new `cdef` statement. It is not available in Python. 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 codes and uses a Python interpreter when they are called from Python. Essentially, such functions have two interfaces:
  - When they are called as a C function
  - When they are called through the Python wrapper

During passing argument from `cdef` to `def` functions and vice versa, there is an automatic type casting occurs:
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. The first step is to take the usual Python file "*.py" and change the extension to* ".pyx".
2. The second and most powerful step that can be used while using Cython is to 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. In the 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. The Next Step is to build your code. The 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 setup.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 at 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 Standard 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"; therefore, it requires such a separate definition in such form.

It is possible to declare and call any C library functions 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
# Dependencies:
#  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.integrate_f(0.0,100.0,100000)
integration.integrate_f_std(0.0,100.0,100000)
```

Use Python functools package

Functools provide wrappers to functions with a memoizing trick from dynamic programming. For example, @functools.lru_cache(maxsize) saves up to the maxsize most recent calls. It can save time if the function is periodically called with the same arguments.

Profiling Python Code with Python Tools

Python interpreter has built-in profiling tools: cProfile and profile. Invocation of these profiling tools for your code snippet can be done in the following way. Example:

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

# Source: sum_with_numpy.py
import numpy as np

for i in range(10000):
    z = np.arange(100000).sum()
```

To launch the Python profile, you should invoke the following:

```
python -m cProfile sum_with_numpy.py
```

or

```
python -m profile sum_with_numpy.py
```

Information about the text report format, which is dumped into the standard output, can be read from python profile module documentation. The main values are the following:

- **ncalls** -- the number of calls.
- **tottime** -- the total time spent in the given function, excluding time made in calls to sub-functions.
- **cumtime** -- is the cumulative time spent in this and all subfunctions.

For most use cases, the most prevalent way to measure performance is using the cProfile module.

Another built-in tool for profiling small code snippets from Python interpreters is timeit. It can be invoked like this and used for measuring the time for the execution of Python statements from several repeating trials:

```
python -m timeit --number 200 --setup "" --unit=sec "'-'.join([str(n) for n in
range(1)])"
```

The timeit can be used not only as a separate tool but can be used inside the Python code itself:

```
import timeit
#....
timeit.timeit("'-'.join([str(n) for n in range(1)])", setup="", number=1000000)
```

Profiling Python Process with Tools Available in Operating Systems: Windows OS

From the perspective of the Operating System (OS), the Python Process is just a program operating in user space. In modern OS, the processes cannot issue direct requests to BIOS, and all communication with OS happens via System Calls. So, even though your program is not built from some source code:

- You are using a precompiled/prebuilt version of the Python interpreter
- You are using a precompiled/prebuilt version of dynamic libraries distributed with packages
- You also have the source code of your Python program that is interpreted by the interpreter on the fly.

SysInternals Suite from Mark Rusinovich et al

To get a general picture of the executed Python process with your scripts, one way is to observe it from system call characteristics. To take hands-on experience and also analyze overheads from Python, for example, you can use the following code snippet:

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

# filename: test.py

import numpy as np
import time

s = time.time()
for i in range(10000):
    z = np.arange(100000).sum()
    if i == 10000-1:
        print("Sum: ", z)
e = time.time()
print("Elapsed Time: ", (e-s)*1000, "milliseconds")

input()
```

Hot to run:

```
python3 test.py
```

Only if you wish to compare C++ and Python code can you can create equivalent code for your Python Logic like the following code:

```
#include <iostream>
#include <chrono>
#include <iterator>
// #include <windows.h>
```

```
namespace chrono = std::chrono;

int main()
{
    auto start = chrono::steady_clock::now();
    for (size_t i = 0; i < 10000; ++i)
    {
        int32_t sum = 0;
        for (int32_t j = 0; j < 100000; ++j)
            sum += j;

        if (i == 10000 -1)
            std::cout << "Sum: " << sum << "\n";
    }
    auto end = chrono::steady_clock::now();

    std::cout << "Eclapsed Time: " << chrono::duration_cast<chrono::milliseconds>
(end - start).count() << " milliseconds\n";

    //void * pp =VirtualAlloc(0, 1024 * 1024 * 100, MEM_RESERVE, PAGE_READWRITE);
    //void* pp2 = VirtualAlloc(0, 1024 * 1024 * 400, MEM_RESERVE|MEM_COMMIT,
PAGE_READWRITE);

    getchar();
    return 0;
}
```

Unfortunately, to build even a simple C++ Program under Windows OS you have to install Microsoft Visual Studio or Microsoft Visual Studio C++ Command Line Tools.

Hot to build and run:

```
build_and_run.bat
```

Content of `build_and_run.bat`:

```
:: Even to build simple code snippets, you should have a collection of programs
(toolchain) that will be used together to build various applications for OS
::  https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-line?
view=msvc-170

:: For compiler flags references:
::   https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options-
listed-alphabetically?view=msvc-170
::   https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options-
listed-by-category?view=msvc-170

:: Execute vcvarsall.bat from Windows SDK or Visual Studio.
:: Default Path for Visual Studio 2022
call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat" x86_amd64

echo **************IMPORTING MSVC TOOLCHAIN IS
FINISHED**********************************************
```

```
:: Compile and link test.cpp to executable
cl.exe /std:c++latest /EHsc /MT /Ot /O2 /GL test.cpp
echo ***************COMPILING/LINK IS
FINISHED***********************************************************

test.exe
```

At the end of the script, there is an `input()` which will wait for input, and similar to C++ code there is a blocking for waiting input from standard input and processes will be alive. Sometimes it can be helpful for profiling to prevent the Python process from dying early.

For Windows OS, collecting a large number of counters is possible via the SysInternals Suite created by Mark Russinovich:

- Process Monitor will allow you to collect statistics and systems call information (with passed argument) that process (such as `python.exe`) exhibits to work with Files, Registry, Network, Dynamic Libraries, Threads, and Processes. It also allows us to inspect how deep the call stacks are and inspect them during the timeline of execution where there is a bottleneck - I/O, Memory, and CPU Computing.
- Process Explorer. It allows inspecting the name of files that are currently open to specific processes (such as `python.exe`) and a list of dynamic libraries (`.dll`) mapped into virtual images of the process (such as `python.exe`).
- VMMap. It demonstrates a process virtual memory map.
- RAMMap. It demonstrates a distribution of physical DRAM memory among different processes inside Windows OS.

Even though these tools have a nice GUI interface, using them if you have a lack of Operating Systems background may not be easy. These tools are powerful profiling/inspection tools that can be used, for example, to find malware software in the OS or get a general picture of what is going on.

If you have never heard about these tools and the tools look a bit complicated for you, please take a look at some talk by Mark Rusinovich (https://en.wikipedia.org/wiki/Mark_Russinovich). E.g. License to Kill: Malware Hunting with the SysInternals Tools

.

Python is also used by people without a CS background but with another background (biology, chemistry, etc.). The terminology used in these tools has an OS system flavor. Below, we will present some terminology used in these tools:

> **User Space Time** - your Python interpreter, is not the only one thing running inside Windows OS (press CTRL+Esc and you will see it). User space-time is the time that your process (python.exe) has spent in user space, which includes code from python.exe and loaded dynamic/shared libraries. This time excludes the time that the system spends on other processes and the time that was needed to handle the system request inside the OS.
>
> **Kernel Space Time** - your Python interpreter and any program (written in C, C++, Assembly, etc.) will request OS for different utility functionality, for example, to open the file. The amount of this utility functionality is huge and this is why the development of OS is treated as a big thing by itself. At a particular moment, your program initializes the System Call, and you form the request to OS (in fact, to the I/O Dispatcher of OS), and your execution thread that executes code inside the Python interpreter will be typically blocked and will be asleep. In this moment, on behalf of this thread, the OS will spend time executing logic inside the

> kernel services and in the stack of the drivers to handle your request. All these operations that the OS did for your thread after I/O Dispatching to correct the Driver(s) will take some time. All this time collectively is called Kernel Space-Time.
>
> **Working Set** -- *working set* or *pinned memory* or *non-paged memory* is the same concept, but the name of the concept depends on the Operating System and on the background of the speaker. The concept means the actual physical DRAM memory dedicated to your process.
>
> **Virtual Memory** -- this concept separates a program's view of memory from the system's physical memory view. In general, an operating system decides when to store the program's code and data in physical memory and when to store it in some file. Virtual memory is a conceptual memory that you may want to address.
>
> **Commit Memory** -- almost always, the tools from SysInternals report the Committed Virtual Memory that your process has requested. The *commit limit* is the sum of physical memory and the sizes of the paging files used for backing up your data. In Windows OS, there is another type of memory *"reserved virtual memory, which is a memory address reserved by the application, but the application does not (and can not) use it until memory is committed. When a process commits a region of virtual memory, the operating system guarantees that it can maintain all the data the process stores in the memory either in physical memory or on disk.
>
> **Physical Memory** - The installed physical memory in the computer by the Windows memory manager populates memory with the code and data from all processes, device drivers, and the OS. The amount of memory can affect performance because when data or code a process or the operating system needs is not present, the memory manager must bring it in from disk.
>
> **System Virtual Memory Limit** - the physical memory size plus the maximum configured size of any paging files.
>
> **Nonpaged Memory (Kernel) Pool** - Authors of drivers for Windows OS have two options for selecting the place from which to allocate the memory. One of these resources is a nonpaged memory pool. The OS kernel and drivers use a nonpaged pool to store data that might be accessed when the system can't handle page faults. The kernel enters such a state when it executes interrupt service routines (ISR) and deferred procedure calls (DPC). A nonpaged pool is always kept present in physical memory. Information about this value can be obtained from *"Process Explorer->System Information->Memory"*.
>
> **Paged (Kernel) Pool** - A paged pool is used by OS and device drivers in situations that can store data that is backed in the paging file, and not necessarily presented in physical memory. Information about this value can be obtained from *"Process Explorer->System Information->Memory"*.
>
> **Context Switches** - When the time slice has elapsed for the thread assigned to a specific processor, a running thread needs to wait, or a thread with a higher priority has become ready to run, and the OS performs a context switch. A high-level picture is that the OS saves the context of the thread that just finished executing, places the thread that just finished executing at the end of the queue for its priority, and finds the highest priority queue that contains ready threads and executes it. After this, the OS dispatches another thread ready to be executed in the processor core.

Process Monitor allows you to capture your application (as a Python interpreter process named `python.exe`) and obtain the following statistics:

- Windows Registry Access Statistics
- File Summary Statistics
- Process Activity Timelines
- Amount of sent and received bytes via Communication Network
- Stack Summary. That gives you the view of stacks.

As we have mentioned, Process Monitor not only produces statistics but also produces pretty detailed information about system calls devoted to Registry, Filesystem, Network, Creation of Threads inside your Python interpreter, and Load Dynamic Libraries. Also, Process Monitor not only detects one of these event types, but also provides you with a calling stack at the moment of the call, arguments for a call (such as the name of the file in case of file activity), and exports this information into Comma Separated Values (CSV) text file, that can be opened e.g. in Microsoft Office tools such as Microsoft Excel.

In addition to getting aggregated statistics about some events, the *Tools->Count Occurrences* can be obtained.

For example during the launching of this Python code in an interactive session:

```
import numpy
```

- The path: `C:\Program Files\python3.9\DLLs` has been used `507` times.
- In total it was `575` accesses to Windows Registry
- In total it was `7982` accesses to File System
- Number of sent and received bytes across the network: `0`
- From the analysis of operations duration the most time is spent on File I/O access, which unfortunately has no relation to `numpy` itself. Rather than that, these files are dynamic libraries that are required to support the Python interpreter by itself.
- The interactive session during its run has used `27` Megabytes of physical DRAM.
- The interpreter first carried approximately `4500` file operations, and after that, it carried `3400` file operations.
- Interactive session during its run has used `500` Megabytes of virtual committed memory. The pretty horrible aspect is that this memory, once Numpy has been imported, is not freed in any way. This virtual memory is committed by the process.

If we open VMMap, it's possible to observe that after `import numpy`, the 66 MBytes is a memory dedicated to images (i.e., executable file `python.exe` interpreter and loaded dynamic libraries). The committed 500 Megabytes are private data specific for your interpreter, for `numpy`, and for libraries under which the Python interpreter and `numpy` depend.

Several other tools from the SysInternals suite can be helpful as well:

- pskill. Windows OS does not come with a command-line 'kill' utility. The `pskill` can be used to kill processes by their identifier or by the name of the executable binary image. Example: `pskill python.exe` kills all Python interpreters running in the current system.
- pslist. These tools obtain as input the name of the process image file and reports:
  - Number of Threads (Thd)
  - Number of Handles to kernel objects(Hnd)

    - Virtual Memory in KBytes (VM)
    - Working Set in KBytes (WS)
    - Page Faults (Faults)
    - Non-Paged Pool (NonP)
    - Paged Pool (Page)
    - Context Switches (Cswtch).

Example: `pslist -x python`

- listdlls. ListDLLs is a utility that reports the Dynamic Link Library(DLL) loaded into processes or lists the processes that have a particular DLL loaded.

  Example:

  ```
  Listdlls.exe python.exe
  ```

Understand which underlying Dynamic Libraries are loaded into the Python interpreter

Next, we want to highlight how to understand what binaries implement specific functionality. It can provide information for you about versions, names, and design choices for the implementation of specific module functionality. The Python interpreter is a binary application working in the userspace of the OS. At its core, the Python interpreter interprets commands.

The big extra functionality is obtained from using Python Modules, which as said in Python Glossary can be one of two types:

- Pure Module written in Python
- Extension Module written in low-level (in Python terminology) C/C++/Java/etc.

In the case of using modules written in compiled languages, they will be loaded in the form of a shared dynamic library object (.so) file for Linux or as (.dll) for Windows. However, this dynamic library has a `.pyd` file extension.

> In terms of flexibility, the pure modules are a good design for creating software, but whether such design is good or bad from the general perspective of creating software (especially when computation time is important) is out of the scope of this note.

For example, to identify which Dynamic Libraries implement `numpy` functionality, you can do the following:

1. Open Python interpreter: `python`
2. Dump the list of used dynamic libraries in the python process with Listdlls: `listdlls.exe python.exe -v > a.txt`
3. Import numpy in python interpreter: `import numpy`
4. Dump the updated list of used dynamic libraries in the python process with Listdlls: `listdlls.exe python.exe -v > b.txt`
5. Use one of the diff tools (Araxis Merge or GNU Diff in Windows OS available as a part of MSYS2) to compare a.txt and b.txt: `diff a.txt b.txt`.

While doing this in my Machine with installed `Numpy 1.21.4` I can observe that one of the load libraries is `openBLAS`. From this, we can conclude that Numpy, in its implementation, leverages this library. If you want to look into the name of symbols of specific dynamic libraries under Windows it's possible to use dll export viewer. If you plan to automatize this process and write a script in Windows Batch, Bash, or Python, you should use the dumpbin tool from Microsoft MSVC Toolchain. For example, in the following way:

- Activate Environment from Windows Command Line:

```
%comspec% /k "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat" x86_amd64
```

- List of symbols:

```
dumpbin /exports "C:\Program Files\python3.9\lib\site-packages\numpy\.libs\libopenblas.XWYDX2IKJW2NMTWSFYNGFUWKQU3LYTCZ.gfortran-win_amd64.dll"
```

- Module preferred image base address:

```
dumpbin /headers "C:\Program Files\python3.9\lib\site-packages\numpy\.libs\libopenblas.XWYDX2IKJW2NMTWSFYNGFUWKQU3LYTCZ.gfortran-win_amd64.dll"  | grep "image base"
```

- Target compute architecture:

```
dumpbin /headers "C:\Program Files\python3.9\lib\site-packages\numpy\.libs\libopenblas.XWYDX2IKJW2NMTWSFYNGFUWKQU3LYTCZ.gfortran-win_amd64.dll" | grep headers
```

Profiling Python Process with Tools Available in Operating Systems: Linux OS

The close-by concepts in terms of userspace application are presented in Linux OS with slightly changed terminology:

> **Two most important types of memory for profiling applications:**
>
> - Current resident set size (VmRSS) - is analogous to Windows OS Working set and represents allocated memory from DRAM for your application and is not swapped out to the disk. So it only includes the portion of the memory that is currently resident in physical RAM.
> - Current virtual memory size (VmSize) - is currently allocated virtual memory for your process. It includes all the memory that the process can access, including both the physical RAM and swap space.

To obtain information about these metrics in Linux OS, firstly you should launch the interpreter:

```
import numpy
```

Next, to obtain information about memory consumed by this process, if your Python interpreter isthe only one Python interpreter currently running in the system, then just call:

```
cat /proc/$(pidof python)/status
```

To find out which shared libraries a process is currently using, we can list the contents of the corresponding Linux-specific files you can make:

```
cat /proc/$(pidof python)/maps
```

This file demonstrated the memory segments and libraries mapped by a program. There is no explicit information about the size of mapped images/libraries. Each line corresponding at the beginning contains a pair of hyphen-separated numbers indicating the virtual address range (in hexadecimal format) at which the memory segment is mapped into the address space of the process (python interpreter).

In Linux OS the OS concepts by design are tried to be represented as files. Dynamic (Shared) Libraries are also files. You can obtain a list of all open files with the lsof command:

```
lsof -p `pidof python`
```

The lsof command can also be used to look into open Internet network connections in the system:

```
lsof -i
```

The analog program to Process Monitor under Linux OS is the system utility `strace`. It allows monitoring system calls. For example, you call it in the following way:

```
strace python -c "import numpy" 2>&1 | grep .so | grep blas
```

The `strace` by itself outputs the name of called system functions with arguments. String arguments are displayed in human-readable form, bitwise flags sometimes are printed as integer constants, sometimes as named constants. Each line ends up with the `=` sign in output (by default output stream for `strace` is stderr). After the equal sign, there is a return code. The code `0` means that all is okay with the completion of the function.

Next some statistics of system calls can be obtained via utilizing `-c` flags of `strace`:

```
strace -c python -c "import numpy" 2>&1
```

The statistics include:

- The number of errors from system calls.
- Number of calls of specific system calls.
- Spend seconds for all system calls of a specific type.
- Percentage of all-time budget spent for all system calls for this call.

Once you e.g. have identified that `import numpy` loads the following shared library `/usr/lib/x86_64-linux-gnu/libblas.so.3`, it is possible to make several things:

- You can view the dependencies for a shared library (or an executable file) via `ldd` system util. It will show dependencies on other `lib*.so` dynamic libraries. Very often, you can see the following dependencies:
  - **libc.so.6** - is the standard library of three things at the same time: C language functions (ISO C11), implementation of POSIX.1-2008, and another OS specific APIs.
  - **ld-linux.so.2** - dynamic linking library for ELF programs.

  Example:

  ```
  ldd /usr/lib/x86_64-linux-gnu/libblas.so.3
  ```

- Next, you can inspect what symbols are actually in some library. For example, with the following command, you can observe that libc.so.6 is not only the C runtime library:

  ```
  objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep -E
  "ioctl|pthread_exit|malloc$"
  ```

- You can list the imported symbols by binary/ dynamic library in the following way:

  ```
  objdump -T <file>|grep "\*UND\*"
  ```

- You can view a summary of the available sections in the binary program. Example:

  ```
  objdump -h /usr/bin/python3
  ```

> The objdump has the following output format:
>
> **1-st** column - the symbol's value, sometimes referred to as its address.
>
> **2-nd** column - flags.
>
> - 'l' means symbols visible only within a particular file being linked (local scope).
> - 'g' means symbols that can be referenced from within functions located in other files (global scope)
> - 'd' debugging symbol
> - 'D' dynamic symbol
> - 'F' symbol is the name of the function
> - 'O' symbol is the name of the object file
>
> **3-rd** column can be represented in several options for a symbol:
>
> - It is the name of the section in which the symbol is defined
> - It has the name *ABS* if essentially there is no section and the address of Symbol is an absolute address.
> - It has name *UND* if symbol currently is not defined. And will be defined later.
>
> **4-th** column - alignment.
>
> **5-th** column - symbol's name is displayed.

Unlike the executable and dynamic library format in Windows OS (PE format), which contains information about the actual name of the dynamic library for symbol, the Linux ELF format does not contain a name binding to a specific dynamic/shared library name.

About Valgrind Tool for Linux OS

Valgrind is a famous tool used for memory debugging, memory leak detection, and profiling in case of using compiled languages. Valgrind works by running the program on a virtual machine that simulates the CPU and memory. Valgrind supports various Posix platforms but is not available under Windows OS. Once you run a program under Valgrind, it performs extensive checking of memory allocations and memory accesses and provides a report with detailed information.

The Valgrind is a simulator. Once you use it, then in terms of wall clock time, your program (Python interpreter process) will run very slowly. But Valgrind and CPU simulators in general produce accurate and repeatable performance counters.

Valgrind is not only a single tool, but it internally contains several tools (https://valgrind.org/info/tools.html,) and it includes:

- Memcheck - Detects memory-management problems.
- Callgrind - Call functions profiler and CPU cache profiler.
- Massif - Heap profiler.
- Helgrind - Thread debugger that finds data races in multithreaded programs.
- Drd - Tool for detecting errors in multithreaded C/C++ programs.

To install Valgrind in Linux OS with apt package manager:

```
sudo apt-get install valgrind
```

Callgrind

For example, you can analyze how the running code is using CPU Caching in the emulated environment:

```
valgrind --tool=callgrind --simulate-cache=yes python -c "import numpy"
```

Callgrind measures only the code that is executed. Please be sure you are making diverse and representative runs that exercise all appropriate code paths. Also, callgrind records the count of instructions, not the actual time spent in a function. Finally, the costs associated with I/O won't show up in the profile.

The result of this program returns the following counters:

- **Ir:** Instructions executed
- **I1mr:** Instruction L1 cache read misses (instruction wasn't in L1 cache but was in L2)
- **I2mr:** Instruction L2 cache read misses (instruction wasn't in L1 or L2 cache, and had to be fetched from memory)
- **Dr:** Total memory reads
- **D1mr:** Data L1 cache read misses (data was not in L1 data cache, but in L2 data cache)
- **D2mr:** Data L2 cache read misses (data was not in L1 or L2 data cache)

- **Dw:** Total memory writes
- **D1mw:** D1 cache write misses (location not in L1 cache, but in L2)
- **D2mw:** L2 cache data write misses

Please take in mind:

- L1 miss will typically cost around 5-10 cycles
- L2 miss can cost as much as 100-200 cycles

Massif

With massif you can look into dynamic memory allocation happening in the program. Any executable code operating in Linux OS, in case of dynamically allocated memory, utilizes one of the following functions:

- Use C/C++ runtime memory allocations with malloc, calloc, realloc, new, new[]
- Use lower-level OS system calls such as mmap, mremap, brk, memalign.

Valgrind can measure all these calls in the following way:

```
rm massif.out*
valgrind --tool=massif --pages-as-heap=yes python -c "import numpy"
```

To visualize results in GUI, you can use massif-visualizer, and to visualize results in the terminal, you can use `ms_print`:

```
ms_print massif.out.24492
```

To visualize the histogram of consumed Megabytes from Dynamic Memory allocation (in OY axis) as a function of time in milliseconds (in OX axis) you look at the beginning of the report:

```
ms_print massif.out.24492 | head -n 35
```

Helgrind

To use this tool, use the following format:

```
valgrind --tool=helgrind python -c "import numpy"
```

Helgrind is a tool for detecting synchronization errors in programs that use the POSIX pthreads threading primitives. Helgrind can detect:

- Misuses of the POSIX pthreads API.
- Potential deadlocks arising from lock ordering problems.
- Accessing memory without adequate locking or synchronization (Data races)

HeapTrack. A heap memory profiler for Linux

An alternative memory profiling tool for Linux is https://github.com/KDE/heaptrack
It has a lower overhead than Valgrind and nice built-in GUI visualizers.

Profiling Hardware Counters with Perf Tool

To install the Perf tool under Linux, you should perform the following command mentioned inthe Ubuntu questions website how-to-install-perf-monitoring-tool:

```
apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r`
```

In modern CPUs from ARM Cortex A5 link, most modern processes support the Performance Monitoring Unit (PMU) unit. There is a helper library libpfm4 available in Linux OS that collects all the needed information from this unit. The Library libpfm4 saves all counters and carries collecting of them.

This library is exploited by the tool called perf and this tool measures various hardware counters per process.

> Warning **A**: Most Counters are poorly documented.
> Warning **B**: Maybe observing all counters will require superuser access to the machine.

Very good, organized one-liner commands for perf are available here Homepage of Brendan Gregg.

Some of them:

One liners for perf commands (Original source):

- `perf stat python -d "pass"`. Obtain CPU counter statistics for launching Python interpreter with empty command including page faults, branch misprediction, cache misses, and CPU migration. A percentage in the perf stat output means the percentage of time that the specific event was being measured. However, it does not necessarily indicate a bottleneck. To identify a bottleneck, you need to compare the performance metrics of different components.
- `perf stat python -c "import numpy"`. Obtain CPU counter statistics for launching the Python interpreter and import the numpy library.
- `perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses python -c "import numpy"`. Obtain Translation Lookaside Buffer (TLB) statistics.

  > **Why TLB Cache is important:**
  >
  > The Control Unit inside the microprocessor operates at the level of micro-operations, and it essentially selects the way to connect electrical components using multiplexers and demultiplexers, turning on/off different electronic components and controlling the control lines. The Load Store units (LS) in the execution pipeline are in charge of carrying memory access. These LS units have access to the Register File, the TLB for address translation, and the CPU Memory Cache. If the data is not available in the Cache, then the CPU Cache requests a block of memory from the Memory Controller, which accesses the DRAM memory via the DRAM memory controller.
  >
  > Even if you are from the world of scripting languages and you may miss details about memory access, you may have heard about L1, L2, L3 (Last Level Cache) Caches. However, there is a more horrible thing than "Data Cache Miss" - it is a miss in the Translation Lookaside Buffer (TLB). To read code or data from memory, the first step is to find the actual physical address of the specific memory location. This operation occurs for every instruction of a program. Without the TLB, the virtual addressing mechanism would require several accesses to different page tables, significantly

> increasing the time needed. The TLB is a cache that stores the mapping between virtual page numbers and physical frame numbers, speeding up the address translation process for memory access.

- `sudo perf stat -e 'syscalls:sys_enter_*' python -c "pass"`. This command count system calls. One system call, roughly speaking, takes 100-2000 cycles from the CPU. In Windows OS the switching from userspace to kernel space takes 1000 cycles. So if you have a lot of System Calls, it will hurt your performance.
- `sudo perf stat -e 'block:*' python -c pass`. Count block device I/O events. These events are addressed directly to the block device. These events happen when a block device I/O request is issued (disk I/O).
- `sudo perf stat -e 'ext4:*' python -c "import numpy"`. Count `ext4` filesystem events. If you are not sure about your filesystem, you can observe disk partition sizes and use the filesystem in a partition with the `df` command in POSIX OS. When specifying a file or directory, the section on which the file is located will be shown: `df -Th`.
- `perf stat -e migrations python -c "import numpy"`. Report the number of process migrations. In computing, process migration is a specialized form of process management whereby processes are moved from one computing environment to another.
- `perf list`. List all currently known software and hardware events in the OS.

Partially Ported Sysinternals Software Suite for Linux

Sysinternals utilities help you manage, troubleshoot, and diagnose application usage for both Windows and Linux systems and applications. Demo of usage Sysinternals for Linux deep dive (demo), 2022. At the current moment, several tools have been ported to Linux:

- https://github.com/Sysinternals/ProcMon-for-Linux
- https://github.com/Sysinternals/SysmonForLinux
- https://github.com/Sysinternals/ProcDump-for-Linux
- https://github.com/Sysinternals/SysinternalsEBPF

However, the most useful tool from this Process Monitor for Linux.

Network Information

Please install the following programs:

- For nicstat: `sudo apt-get install nicstat`

#	Command	Description	Requires Root Access
1	`netstat -i`	Interface Counters. **Iface:** interface name, **MTU:** Maximum Transmission Unit, **RX-OK:** Count of *packets* received successfully, **RX-ERR:** Count of receive errors, **RX-DRP:** Count of dropped packets on receive, **TX-OK:** Count of packets transmitted successfully, **TX-ERR:** Count of transmit errors, **TX-DRP:** Count of dropped packets on transmit	No

#	Command	Description	Requires Root Access
2	`ip -s link`	Interface Counters. In Bytes and Packages. **RX:** received, **TX:** Count of packets transmitted successfully.	No
3	`nicstat`	The byte throughput for send and receive through interface. **rKb/s** - read Kbytes/s, **wKb/s** - write Kbytes/s, **rPk/s** and **wPk/s** - read/write Packets per second, **rAvs** and **wAVS** -- read/write average size in bytes.	No

CPU and Memory and Input-Output Information

Please install the following binaries:

- **syscounts** -- `sudo apt-get install libbpf-tools`
- **lscpu** -- `sudo apt install util-linux`
- **lstopo** -- `sudo apt install hwloc`
- **mpstat** -- `sudo apt install sysstat`
- **vfsstat-bpfcc** -- `sudo apt install bpfcc-tools`

#	Command	Description	Requires Root Access
1	`uptime`	Load averages	No
2	`vmstat -S M`	System wide statistics: **swpd** - swaped-out memory(MB), **free** - free memory(MB), **buff** - buffer caches (MB), **us** - user-time percent, **sy** - kernel-time percent, **id** - idle percent, **wa** - I/O blocking percent in disk I/O	No
3	`mpstat -P ALL`	Per-CPU-core balance statistics: **user, system, iowait** times; **irq** - hardware interrupts triggered by external hardware devices, **soft** - software interrupts triggered by software.	No
4	`free -h`	Memory usage including swap file	No
5	`iostat -sxz`	Disk I/O statistics. **tps** - disk transactions/second; **kB/s** - the total data transfer rate both reads and writes combined in kilobytes per second; **rqm/s** - queued requests/second; **await** - average I/O response time (ms), **util** - utilization.	No

#	Command	Description	Requires Root Access
6	`iostat`	Disk I/O statistics	No
7	`sar -n DEV`	Network device statistics	No
8	`syscount-perf -d 1`	System call statistics system-wide	Yes
9	`lscpu`	Information about CPU and CPU Architecture relative information.	No
10	`lstopo --output-format console`	Show hardware topology of the system	No
11	`lsblk -o NAME,MODEL,SIZE,TYPE,ROTA`	List physical HDDs and SSDs in your system.	No
12	`swapon`	Show configured devices for swap files and swap usage	No
13	`sar -B`	Displays memory paging statistics in Linux. Pages read from disk (**pgpgin/s**) and pages written to disk (**pgpgout/s**) per second. If **pgpgout/s** is high, the system might be under memory pressure. The **pgpgin/s** or **pgpgout/s** high values may suggest high swap activity.	No
14	`sar -r`	The `sar -r` is used to display memory statistics.	No
15	`sar -S`	Swap space statistics: **kbswpfree** - amount of free swap space in kilobytes; **kbswpused** - amount of swap space used in kilobytes; **%swpused** - percentage of swap space used.	No
16	`sar -v`	Filesystem statistics: **dentunused** - unused dirs, **file-nr** - file handles in use, **inode-nr** - inodes (file metadata) in use	No
17	`df -h`	Report filesystem usage and available capacity	No
18	`df -h fname`	Report filesystem usage and available capacity for the device which holds fname file.	No

#	Command	Description	Requires Root Access
19	`vfsstat-bpfcc 1 1`	Virtual File System statistics. Virtual File System (VFS) layer is included and always used in Linux (and most modern UNIX-like operating systems) for all file systems (FS). It acts as an abstraction layer between the user-space file operations (opening, reading, writing) and the actual underlying file system (e.g. ext4). Writing directly to a disk without using a filesystem is possible, but it is typically not performed - it almost always breaks the filesystem and requires root privileges.	Yes.

Profiling Python Process with Cross-Platform Tools

GPU-related information with NVIDIA-SMI (Cross-Platform)

#	Command	Description	Requires Root Access
1	`ls -1 /dev/nvidia*`	List of installed NVIDIA Devices in the system	No
2	`nvidia-smi -L`	List of installed NVIDIA Video Cards	No
3	`nvidia-smi -q -d MEMORY`	NVIDIA Devices Memory Report	No
4	`nvidia-smi -q -d UTILIZATION`	NVIDIA Devices Utilization Report	No
5	`nvidia-smi -q -d TEMPERATURE`	NVIDIA Devices Temperature Report	No
6	`nvidia-smi -q -d POWER`	NVIDIA Devices Power Report	No
7	`nvidia-smi -q -d CLOCK`	NVIDIA Devices Clock Report	No

Acknowledgements

Konstantin Burlachenko would like to acknowledge:

- The original author and contributors to the official Python Tutorial.
- Modar Alfadly for providing in-depth PyTorch, NumPy, Python Tutorials during the course Deep Learning for Visual Computing, CS323 with prof. Bernard Ghanem.
- Aleksandar Cvejic for providing suggestions to improve this note with Profiling information.

Contributions to This Document

If you want to contribute to any of the goals above and support further development of the document, we are only open to adding you as an *Author* in case you contributed to creating sections. If you want to clarify and polish the text, provided that you have used Python for at least 2 years we would appreciate polishing from your side, and we would be glad to you as *Editor*.

From our side, we ask you to:

a. Provide short examples (or no examples) to support your point if it is possible.

b. Provide the minimum code snippet. We do not want to see expressive power when you can create a 5-level nested comprehension construction in one line.

c. Please provide links to sources of information and original links to sources or documentation. The knowledge in Society tends to be spread across different communities, rather than one person knowing everything, concise references will help other people go beyond the provided information.

d. For big software packages with which you gained knowledge by spending plenty of time you have two options: Either Create a short Section or create a reference to the original documentation.

References

Introduction Documents

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

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 contents (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 to Python

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

Tutorials for Libraries

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

How To

[10] Python Tutorial http://www.java2s.com/Tutorial/Python/CatalogPython.htm

[11] Open Source book about Python Tips: 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/

Performance Relative Materials

[14] Systems Performance: Enterprise and the Cloud 2nd Edition:
https://www.amazon.com/Systems-Performance-Brendan-Gregg/dp/0136820158/ref=sr_1_1