Chapter 1

Why Do People Use Python?

Software quality

- Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages.
- Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and function programming.
- In the Python way of thinking, **explicit is better than implicit**, and simple is better than complex

Developer productivity

- Python code is typically *one-third to one-fifth* the size of equivalent C++ or Java code.
- Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.
- It is deliberately optimized for *speed of development*—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools.

Program portability

- Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines.

Support libraries

- Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*.
- Python's *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more
- The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

Component integration

- Python has variety of integration mechanisms.
- Such integrations allow Python to be used as a product *customization and extension* tool.

Is Python a "Scripting Language"?

- Python is a general-purpose programming language that is often applied in scripting roles.
- Python is probably better known as a *general-purpose programming language that blends procedural, functional, and object-oriented paradigms.*

What’s the Downside?

- Its *execution speed* may not always be as fast as that of fully compiled and lower-level languages such as C and C++.
- The standard implementations of Python today compile (i.e., translate) source code statements to an intermediate format known as ***byte code*** and then interpret the byte code.
- Byte code provides portability, as it is a platform-independent format.
- Whenever you do something “real” in a Python script, like processing a file or constructing a graphical user interface (GUI), your program will actually run at C speed, since such tasks are immediately dispatched to compiled C code inside the Python interpreter.
- More fundamentally, Python’s speed-of-development gain is often far more important than any speed-of-execution loss, especially given modern computer speeds.

What Can I Do with Python?

Systems Programming:

- Python’s built-in interfaces to operating-system services make it ideal for writing portable, maintainable system administration tools and utilities (sometimes called *shell tools*).

GUIs

- Python’s simplicity and rapid turnaround also make it a good match for graphical user interface programming on the desktop.
- Python comes with a standard object-oriented interface to the Tk GUI API called ***tkinter*** (*Tkinter* in 2.X) that allows python programs to implement portable GUIs with a native look and feel.
- Example: tkinter, PMW, wxPython, Dabo(built on top of wxPython and tkinter), Qt with PyQt/PySide2, MFC with PyWin32, .Net with IronPython, Swing with Jython or JPype

Internet Scripting

- Python comes with standard Internet modules that allow Python programs to perform a wide variety of networking tasks, in client and server modes.

Component Integration

- Python’s ability to be extended by and embedded in C and C++ systems makes it useful as a flexible glue language for scripting the behavior of other systems and components.
- the *Cython* system allows coders to mix Python and C-like code.
- Larger frameworks, such as Python’s *COM* support on Windows, the Jython *Java*-based implementation, and the IronPython .NET-based implementation provide alternative ways to script components.
- On Windows, for example, Python scripts can use frameworks to script Word and Excel, access *Silverlight*, and much more.

Database Programming

- The Python world has also defined a *portable database API* for accessing SQL database systems from Python scripts.

Numeric and Scientific Programming

- Python is also heavily used in numeric programming.
- The ***PyPy*** implementation of Python has also gained traction in the numeric domain, in part because heavily algorithmic code of the sort that’s common in this domain can run dramatically faster in PyPy—often 10X to 100X quicker.

- the ***NumPy*** high-performance numeric programming extension for Python mentioned earlier includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more.

How Does Python Stack Up to Language X?

While other languages are also useful tools to know and use, many people find that
Python:

- Is more powerful than ***Tcl***. Python's strong support for "programming in the large" makes it applicable to the development of larger systems, and its library of application tools is broader.
- Is more readable than ***Perl***. Python has a clear syntax and a simple, coherent design. This in turn makes Python more reusable and maintainable and helps reduce program bugs.
- Is simpler and easier to use than ***Java*** and ***C#***. Python is a scripting language, but Java and C# both inherit much of the complexity and syntax of larger OOP systems languages like C++.
- Is simpler and easier to use than ***C++***. Python code is simpler than the equivalent C++ and often one-third to one-fifth as large, though as a scripting language, Python sometimes serves different roles.
- Is simpler and higher-level than ***C***. Python's detachment from underlying hardware architecture makes code less complex, better structured, and more approachable than C, C++'s progenitor.
- Is more powerful, general-purpose, and cross-platform than ***Visual Basic***. Python is a richer language that is used more widely, and its open source nature means it is not controlled by a single company.
- Is more readable and general-purpose than ***PHP***. Python is used to construct websites too, but it is also applied to nearly every other computer domain, from robotics to movie animation and gaming.
- Is more powerful and general-purpose than ***JavaScript***. Python has a larger toolset and is not as tightly bound to web development. It's also used for scientific modeling, instrumentation, and more.
- Is more readable and established than ***Ruby***. Python syntax is less cluttered, especially in nontrivial code, and its OOP is fully optional for users and projects to which it may not apply.
- Is more mature and broadly focused than ***Lua***. Python's larger feature set and more extensive library support give it a wider scope than Lua, an embedded "glue" language like Tcl.
- Is less esoteric than ***Smalltalk***, ***Lisp***, and ***Prolog***. Python has the dynamic flavor of languages like these, but also has a traditional syntax accessible to both developers and end users of customizable systems.

Questions:

1. What are the main reasons that people choose to use Python?
2. Why might you *not* want to use Python in an application?
3. What can you do with Python?

Chapter 2

How Python Runs Programs

Python Interpreter

- it's also a software package called an ***interpreter****.*
- An interpreter is a kind of program that executes other programs.
- the interpreter is a layer of software logic between your code and the computer hardware on your machine.
- Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or something else.

Program Execution

Python's View:

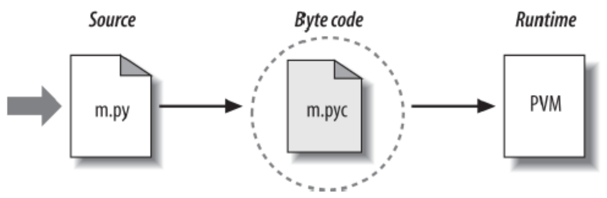
- You type code into text files, and you run those files through the interpreter.
- Specifically, it's first compiled to something called "byte code" and then routed to something called a "virtual machine."

Byte code compilation

- when you execute a program Python first compiles your *source code* (the statements in your file) into a format known as ***byte code***.
- Compilation is simply a translation step, and byte code is a lower-level, **platform-independent** representation of your source code.
- This byte code translation is performed to speed execution —byte code can be run much more quickly than the original source code statements in your text file.
- it will store the byte code of your programs in files that end with a ***.pyc*** **extension** (".pyc" means compiled ".py" source).
- In 3.2 and later, Python instead saves its *.pyc* **byte code files** in a subdirectory named ***__pycache__*** located in the directory where your source files reside, and in files whose names identify the Python version that created them.
- The new *__pycache__* subdirectory helps to avoid clutter, and the new naming convention for byte code files prevents different Python versions installed on the same computer from overwriting each other's saved byte code.
- It works like this:
    - ***Source changes***: Python automatically checks the last-modified timestamps of source and byte code files to know when it must recompile.
    - ***Python versions***: Imports also check to see if the file must be recompiled because it was created by a different Python version, using either a "magic" version number in the byte code file itself in 3.2 and earlier, or the information present in byte code filenames in 3.2 and later.
- The result is that both source code changes and differing Python version numbers will trigger a new byte code file.
- If Python cannot write the byte code files to your machine, your program still works—the byte code is generated in memory and simply discarded on program exit.
- **Python is happy to run a program if all it can find are *.pyc* files, even if the original *.py* source files are absent.** ("Frozen Binaries")
- keep in mind that byte code is saved in files only for files that are *imported*, **not for the top-level files** of a program that are only run as scripts. **it's an import optimization**.

The Python Virtual Machine (PVM)

- Once your program has been compiled to byte code (or the byte code has been loaded from existing *.pyc* files), it is shipped off for execution to something generally known as the **Python Virtual Machine (PVM).**



- the PVM is just a big code loop that iterates through your byte code instructions, one by one, to carry out their operations.
- The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts.
- Keep in mind that all this complexity is deliberately hidden from Python programmers.

Performance implications

- the PVM loop, not the CPU chip, still must interpret the byte code, and byte code instructions require more work than CPU instructions.
- On the other hand, unlike in classic interpreters, there is still an internal compile step.
- The net effect is that pure Python code runs at speeds somewhere between those of a traditional compiled language and a traditional interpreted language.

Execution Model Variations

Python Implementation Alternatives

- five implementations of the Python language—***CPython*, *Jython*, *IronPython*, *Stackless*, and *PyPy*.**
- ***CPython*** is the standard implementation, and the system that most readers will wish to use.

CPython: The standard

- The original, and standard, implementation of Python is usually called **CPython.**
- it is coded in portable ANSI C language code.

Jython: Python for Java

- The **Jython system** (originally known as **JPython**) is an alternative implementation of the Python language, targeted for integration with the **Java programming language**.
- **Jython** consists of Java classes that compile Python source code to **Java byte code** and then route the resulting byte code to the **Java Virtual Machine** (JVM).
- Jython's goal is to allow Python code to script Java applications.
- Jython is slower and less robust than CPython
- Jython's website *http://jython.org* for more details

IronPython: Python for .NET

- **IronPython** is designed to allow Python programs to integrate with applications coded to work with **Microsoft's .NET Framework** for Windows, as well as the Mono open source equivalent for Linux.
- By implementation, IronPython is very much like Jython.
- For more details, consult *http://ironpython.net*

Stackless: Python for concurrency

- the ***Stackless*** Python system is an enhanced version and reimplementation of the standard **CPython** language oriented toward ***concurrency***.

- Among other things, the *microthreads* that Stackless adds to Python are an efficient and lightweight alternative to Python's standard multitasking tools such as threads and processes, and promise better program structure, more readable code, and increased programmer productivity.
- CCP Games, the creator of *EVE Online*, is a well-known Stackless Python user
- Try *http://stackless.com* for more information.

PyPy: Python for speed

- focused on *performance*.
- It provides a fast Python implementation with a *JIT* (just-in-time) compiler

Frozen Binaries

- simply a way to generate standalone binary executables from their Python programs.
- This is more a packaging and shipping idea than an execution-flow concept, but it's somewhat related.
- It is possible to turn your Python programs into true executables, known as ***frozen binaries*** in the Python world.
- Frozen binaries **bundle together the byte code** of your program files, along with the **PVM** (interpreter) and **any Python support files** your program needs, into a **single package**.
- Frozen binaries are not the same as the output of a true compiler—they run byte code through a virtual machine.
- Frozen binaries are also not generally small (they contain a PVM).

Questions

1. What is the Python interpreter?
2. What is source code?
3. What is byte code?
4. What is the PVM?
5. Name two or more variations on Python's standard execution model.
6. How are CPython, Jython, and IronPython different?
7. What are Stackless and PyPy?

****

Download Python: https://www.python.org/downloads/

Creation of virtual environments

- The virtual environment allows us to create a separate virtual environment. You can only install selected Python modules you want in your virtual environment. This makes your environment cleaner and simpler.

Steps to create a virtual env:

- Make a directory for your virtual environment.
- **pip install virtualenv**
- **virtualenv –version**
- **virtualenv myenv** or **python -m venv "D:\@Study\Python\#venv\test1"**
    - This will create a directory myenv along with directories inside it containing a copy of the Python interpreter, the standard library, and various supporting files.
    - A virtual Python environment has a similar directory structure to a global Python installation.
    - **bin** directory contains executables for the virtual environment.
    - the **include** directory is linked to the global Python installation header files.
    - the **lib** directory is a copy of the global Python installation libraries and where packages for the virtual environment are installed
    - the **shared** directory is used to place shared Python packages.
    - **myenv>Lib>site-packages**: site-packages director will all the essential tools. Eg. pip, wheel etc. All the modules which you will install in this environment, will install in this folder and these packages will be isolated from other environment even global python installation.
- **myenv\Scripts\activate.bat**: This command will activate environment.
- **deactivate**: To deactivate the environment

Ref:

- https://docs.python.org/3/tutorial/venv.html
- Optional advance tool: https://docs.python-guide.org/dev/virtualenvs/#virtualenvwrapper