1.1 Getting Started

composingprograms.com/pages/11-getting-started.html

Chapter 1: Building Abstractions with Functions

Computer science is a tremendously broad academic discipline. The areas of globally distributed systems, artificial intelligence, robotics, graphics, security, scientific computing, computer architecture, and dozens of emerging sub-fields all expand with new techniques and discoveries every year. The rapid progress of computer science has left few aspects of human life unaffected. Commerce, communication, science, art, leisure, and politics have all been reinvented as computational domains.

The high productivity of computer science is only possible because the discipline is built upon an elegant and powerful set of fundamental ideas. All computing begins with representing information, specifying logic to process it, and designing abstractions that manage the complexity of that logic. Mastering these fundamentals will require us to understand precisely how computers interpret computer programs and carry out computational processes.

These fundamental ideas have long been taught using the classic textbook *Structure and Interpretation of Computer Programs* (SICP) by Harold Abelson and Gerald Jay Sussman with Julie Sussman. This text borrows heavily from that textbook, which the original authors have kindly licensed for adaptation and reuse under a Creative Commons license. These notes are published under the Creative Commons attribution non-commericial share-alike license version 3.

1.1.1 Programming in Python

> A language isn't something you learn so much as something you join.
>
> —Arika Okrent

In order to define computational processes, we need a programming language; preferably one that many humans and a great variety of computers can all understand. In this text, we will work primarily with the Python language.

Python is a widely used programming language that has recruited enthusiasts from many professions: web programmers, game engineers, scientists, academics, and even designers of new programming languages. When you learn Python, you join a million-person-strong community of developers. Developer communities are tremendously important institutions: members help each other solve problems, share their projects and experiences, and collectively develop software and tools. Dedicated members often achieve celebrity and widespread esteem for their contributions.

The Python language itself is the product of a large volunteer community that prides itself on the diversity of its contributors. The language was conceived and first implemented by Guido van Rossum in the late 1980's. The first chapter of his Python 3 Tutorial explains why Python is so popular, among the many languages available today.

Python excels as an instructional language because, throughout its history, Python's developers have emphasized the human interpretability of Python code, reinforced by the Zen of Python guiding principles of beauty, simplicity, and readability. Python is particularly appropriate for this text because its broad set of features support a variety of different programming styles, which we will explore. While there is no single way to program in Python, there are a set of conventions shared across the developer community that facilitate reading, understanding, and extending existing programs. Python's combination of great flexibility and accessibility allows students to explore many programming paradigms, and then apply their newly acquired knowledge to thousands of ongoing projects.

These notes maintain the spirit of SICP by introducing the features of Python in step with techniques for abstraction and a rigorous model of computation. In addition, these notes provide a practical introduction to Python programming, including some advanced language features and illustrative examples. Increasing your facility with Python should come naturally as you progress through the text.

The best way to get started programming in Python is to interact with the interpreter directly. This section describes how to install Python 3, initiate an interactive session with the interpreter, and start programming.

1.1.2 Installing Python 3

As with all great software, Python has many versions. This text will use the most recent stable version of Python 3. Many computers have older versions of Python installed already, such as Python 2.7, but those will not match the descriptions in this text. You should be able to use any computer, but expect to install Python 3. (Don't worry, Python is free.)

You can download Python 3 from the *Python downloads* page by clicking on the version that begins with 3 (not 2). Follow the instructions of the installer to complete installation.

For further guidance, try these video tutorials on Windows installation and Mac installation of Python 3, created by Julia Oh.

1.1.3 Interactive Sessions

In an interactive Python session, you type some Python *code* after the *prompt*, >>>. The Python *interpreter* reads and executes what you type, carrying out your various commands.

To start an interactive session, run the Python 3 application. Type `python3` at a terminal prompt (Mac/Unix/Linux) or open the Python 3 application in Windows.

If you see the Python prompt, `>>>`, then you have successfully started an interactive session. These notes depict example interactions using the prompt, followed by some input.

```
>>> 2 + 2
4
```

Interactive controls. Each session keeps a history of what you have typed. To access that history, press `<Control>-P` (previous) and `<Control>-N` (next). `<Control>-D` exits a session, which discards this history. Up and down arrows also cycle through history on some systems.

1.1.4 First Example

> And, as imagination bodies forth
>
> The forms of things to unknown, and the poet's pen
>
> Turns them to shapes, and gives to airy nothing
>
> A local habitation and a name.
>
> —William Shakespeare, A Midsummer-Night's Dream

To give Python a proper introduction, we will begin with an example that uses several language features. In the next section, we will start from scratch and build up the language piece by piece. Think of this section as a sneak preview of features to come.

Python has built-in support for a wide range of common programming activities, such as manipulating text, displaying graphics, and communicating over the Internet. The line of Python code

```
>>> from urllib.request import urlopen
```

is an `import` statement that loads functionality for accessing data on the Internet. In particular, it makes available a function called `urlopen`, which can access the content at a uniform resource locator (URL), a location of something on the Internet.

Statements & Expressions. Python code consists of expressions and statements. Broadly, computer programs consist of instructions to either

1. Compute some value
2. Carry out some action

Statements typically describe actions. When the Python interpreter executes a statement, it carries out the corresponding action. On the other hand, expressions typically describe computations. When Python evaluates an expression, it computes the value of that expression. This chapter introduces several types of statements and expressions.

The assignment statement

```
>>> shakespeare = urlopen('http://composingprograms.com/shakespeare.txt')
```

associates the name `shakespeare` with the value of the expression that follows =. That expression applies the `urlopen` function to a URL that contains the complete text of William Shakespeare's 37 plays, all in a single text document.

Functions. Functions encapsulate logic that manipulates data. `urlopen` is a function. A web address is a piece of data, and the text of Shakespeare's plays is another. The process by which the former leads to the latter may be complex, but we can apply that process using only a simple expression because that complexity is tucked away within a function. Functions are the primary topic of this chapter.

Another assignment statement

```
>>> words = set(shakespeare.read().decode().split())
```

associates the name `words` to the set of all unique words that appear in Shakespeare's plays, all 33,721 of them. The chain of commands to `read`, `decode`, and `split`, each operate on an intermediate computational entity: we `read` the data from the opened URL, then `decode` the data into text, and finally `split` the text into words. All of those words are placed in a `set`.

Objects. A `set` is a type of object, one that supports set operations like computing intersections and membership. An object seamlessly bundles together data and the logic that manipulates that data, in a way that manages the complexity of both. Objects are the primary topic of Chapter 2. Finally, the expression

```
>>> {w for w in words if len(w) == 6 and w[::-1] in words}
{'redder', 'drawer', 'reward', 'diaper', 'repaid'}
```

is a compound expression that evaluates to the set of all Shakespearian words that are simultaneously a word spelled in reverse. The cryptic notation `w[::-1]` enumerates each letter in a word, but the `-1` dictates to step backwards. When you enter an expression in an interactive session, Python prints its value on the following line.

Interpreters. Evaluating compound expressions requires a precise procedure that interprets code in a predictable way. A program that implements such a procedure, evaluating compound expressions, is called an interpreter. The design and implementation of interpreters is the primary topic of Chapter 3.

When compared with other computer programs, interpreters for programming languages are unique in their generality. Python was not designed with Shakespeare in mind. However, its great flexibility allowed us to process a large amount of text with only a few statements and expressions.

In the end, we will find that all of these core concepts are closely related: functions are objects, objects are functions, and interpreters are instances of both. However, developing a clear understanding of each of these concepts and their role in organizing code is critical to mastering the art of programming.

1.1.5 Errors

Python is waiting for your command. You are encouraged to experiment with the language, even though you may not yet know its full vocabulary and structure. However, be prepared for errors. While computers are tremendously fast and flexible, they are also extremely rigid. The nature of computers is described in Stanford's introductory course as

> The fundamental equation of computers is:
>
> `computer = powerful + stupid`
>
> Computers are very powerful, looking at volumes of data very quickly. Computers can perform billions of operations per second, where each operation is pretty simple.
>
> Computers are also shockingly stupid and fragile. The operations that they can do are extremely rigid, simple, and mechanical. The computer lacks anything like real insight ... it's nothing like the HAL 9000 from the movies. If nothing else, you should not be intimidated by the computer as if it's some sort of brain. It's very mechanical underneath it all.
>
> Programming is about a person using their real insight to build something useful, constructed out of these teeny, simple little operations that the computer can do.
>
> —Francisco Cai and Nick Parlante, Stanford CS101

The rigidity of computers will immediately become apparent as you experiment with the Python interpreter: even the smallest spelling and formatting changes will cause unexpected output and errors.

Learning to interpret errors and diagnose the cause of unexpected errors is called *debugging*. Some guiding principles of debugging are:

1. **Test incrementally**: Every well-written program is composed of small, modular components that can be tested individually. Try out everything you write as soon as possible to identify problems early and gain confidence in your components.
2. **Isolate errors**: An error in the output of a statement can typically be attributed to a particular modular component. When trying to diagnose a problem, trace the error to the smallest fragment of code you can before trying to correct it.

3. **Check your assumptions**: Interpreters do carry out your instructions to the letter — no more and no less. Their output is unexpected when the behavior of some code does not match what the programmer believes (or assumes) that behavior to be. Know your assumptions, then focus your debugging effort on verifying that your assumptions actually hold.
4. **Consult others**: You are not alone! If you don't understand an error message, ask a friend, instructor, or search engine. If you have isolated an error, but can't figure out how to correct it, ask someone else to take a look. A lot of valuable programming knowledge is shared in the process of group problem solving.

Incremental testing, modular design, precise assumptions, and teamwork are themes that persist throughout this text. Hopefully, they will also persist throughout your computer science career.

Continue: 1.2 Elements of Programming