Teaching materials

15016406: COMPUTER PROGRAMMING

By

Patcharin Kamsing (Ph.D.)

Department of Aeronautical Engineering and

Commercial Pilot

International Academy of Aviation Industry

King Mongkut's Institute of Technology Ladkrabang

Teaching materials

15016406: COMPUTER PROGRAMMING

By

Patcharin Kamsing (Ph.D.)

Department of Aeronautical Engineering and

Commercial Pilot

International Academy of Aviation Industry

King Mongkut's Institute of Technology Ladkrabang

Abstract

Contents

Course Syllabus

International Academy of Aviation Industry
King Mongkut's Institute of Technology Ladkrabang
Aeronautical Engineering and Commercial Pilot
1/2562

---

1. **Course Number**: 15016406
2. **Course Title**: COMPUTER PROGRAMMING
3. **Course Credit**: 3 (2-2)
4. **Instructor**: Patcharin Kamsing (Ph.D.), E-mail: patcharin.ka@kmitl.ac.th
5. **Condition**

   5.1 Prerequisite: None

   5.2 Corequisite: None

   5.3 Concurrent: None

   5.4 Status (Required/Elective): Required
6. **Hours/Week**: 4 Hours
7. **Course Description**:

   แนวคิดของระบบคอมพิวเตอร์องค์ประกอบของระบบคอมพิวเตอร์การปฏิสัมพันธ์ ระหว่างฮาร์ดแวร์และซอฟต์แวร์ แนวคิดของการประมวลผลข้อมูลแบบอิเล็กทรอนิกส์การออกแบบและขั้นตอนการพัฒนาโปรแกรมการเขียนโปรแกรมด้วยภาษาคอมพิวเตอร์ระดับสูง

   Computer concept, computer components, hardware and software interaction, EDP concepts, program design and development methodology, high-level language programming.
8. **Course Outline**

8.1 Course Learning Outcome (CLO)

CLO-1:

CLO-2:

CLO-3:

CLO-4:

CLO-5:

8.2 Learning Contents

Week	Learning content	Remark
1	Chapter 1 Why we Program - Installing and Using Pythin	Explanation/Laboratory
2	Chapter 1 Why we Program - Installing and Using GitHub	Explanation/Laboratory
3	Chapter 2 Variables, expressions and statements	Explanation/Laboratory
4	Chapter 3 Conditionals	Explanation/Laboratory
5	Chapter 3 Conditionals	Explanation/Report
6	Chapter 4 Loop and Iteration (for loop)	Explanation/Laboratory
7	Chapter 4 Loop and Iteration (While loop)	Explanation/Report
8	Midterm Examination	-
9	Chapter 5 Arrays	Explanation/Laboratory
10	Chapter 5 Arrays	Explanation/Laboratory
11	Chapter 6 Functions	Explanation/Laboratory

12	Chapter 7 Files Handling	Explanation/Laboratory
13	Mini-project Python programming	Explanation/mini project
14	Mini-project Python programming	Explanation/mini project
15	Final Examination	-

8.3 Learning Method

- Explaination

- Using computer for coding a program

8.4 Learning Media

- PDF File

- Website: https://patcharinka.github.io/15016406/

8.5 Evaluation

- Midterm Exam 30%

- Final Exam 40%

- Mini-project 10%

- In class activities (including Homework, class attendance, etc) 20%

9. Reading List and Supplementary Texts

10. Teacher Evaluation

10.1 Method for evaluation

- Post-Test

- Learning outcome

10.2 An improvement from the last teacher evaluation

None

11. Giving a recommendation to students about this course and other topic

- Describe the date and time of this course, date and time to have a consult with the teacher by face to face. Explain the method to submit homework and mini-project and explain the learning plan and condition to give a score in the first class.
- Allow a student to send an email to get a suggestion from the teacher in case, not in-office hours.

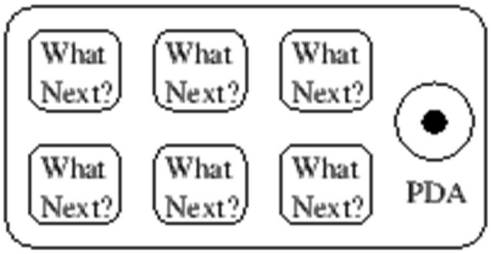
12. The teacher gives this Course Syllabus to student in the first class.

- In the first day (.....), the teacher already gives the course syllabus to the student as shown in the attached document.

Chapter 1 Why we Program

Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem. This teaching material assumes that everyone needs to know how to program and that once you know how to program, you will figure out what you want to do with your newfound skills. [1]

We are surrounded in our daily lives with computers ranging from laptops to cell phones. We can think of these computers as our ``personal assistants'' who can take care of many things on our behalf. The hardware in our current-day computers is essentially built to continuously ask us the question, ``What would you like me to do next?''.



Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite helpful and capable of helping many different things.

Our computers are fast and have vast amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to ``do next''. If we knew this language we could tell the computer to do tasks on our behalf that

were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

This very fact that computers are good at things that humans are not is why you need to become skilled at talking ``computer language''. Once you learn this new language, you can delegate mundane tasks to your partner (the computer), leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

1.1 Installing and Using Python

This teaching material is for basic Python programming which mainly concentrate on Python desktop programming. Therefore, the student needs to install Python environment for themselves. Python installation have two parts namely, 1) Python Editor and 2) Python executes(compiler).

1.1.1 Python IDEs and Code Editor

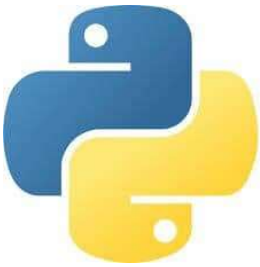
Python is one of the famous high-level programming languages that was developed in 1991. Python is mainly used for server-side web development, development of software, math, scripting, and artificial intelligence. It works on multiple platforms like Windows, Mac, Linux, Raspberry Pi etc. Before exploring more about Python IDE, we must understand what an IDE is. [2]

IDE stands for Integrated Development Environment. IDE is basically a software pack that consist of equipment's which are used for developing and testing the software. A developer throughout SDLC uses many tools like editors, libraries, compiling and testing platforms. IDE helps to automate the task of a developer by reducing manual efforts and combines all the equipment's in a common framework. If IDE is not present, then the developer has to manually do the selections, integrations and deployment process. IDE was basically developed to simplify the SDLC process, by reducing coding and avoiding typing errors. In contrast to the IDE, some developers also prefer Code editors. Code Editor is basically a text editor where a developer can write the code for developing any software. Code editor also allows the developer to save small text files for the code. In comparison to IDE, code editors are fast in operating and have a small size. In fact code editors possess the capability of executing and debugging code.

PyCharm

Spyder

Pydev

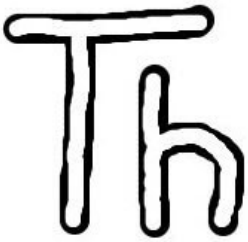


Idle

Wing

Eric Python

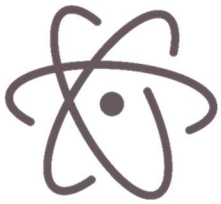
Rodeo



Thonny

Code editors are basically the text editors which are used to edit the source code as per the requirements. These may be integrated or stand-alone applications. As they are monofunctional, they are very faster too. Enlisted below are some of the top code editors which are preferred by the Python developer's world-wide.

Sublime Text



Atom

Vim

Visual Studio Code

In conclusion, IDE is a development environment which provides many features like coding, compiling, debugging, executing, autocomplete, libraries, in one place for the developer's thus making tasks simpler whereas Code editor is a platform for editing and modifying the code only. PyCharm is selected for using in the class since PyCharm is one of the widely used Python IDE which was created by Jet Brains. It is one of the best IDE for Python. PyCharm is all a developer's need for productive Python development.

With PyCharm, the developers can write a neat and maintainable code. It helps to be more productive and gives smart assistance to the developers. It takes care of the routine tasks by saving time and thereby increasing profit accordingly.

Best Features:

1.It comes with an intelligent code editor, smart code navigation, fast and safe refactoring's.

2.PyCharm is integrated with features like debugging, testing, profiling, deployments, remote development and tools of the database.

3.With Python, PyCharm also provides support to python web development frameworks, JavaScript, HTML, CSS, Angular JS and Live edit features.

4.It has a powerful integration with IPython Notebook, python console, and scientific stack.

Pros:

1.It provides a smart platform to the developers who help them when it comes to auto code completion, error detection, quick fixing etc.

2.It provides multiple framework support by increasing a lot of cost-saving factors.

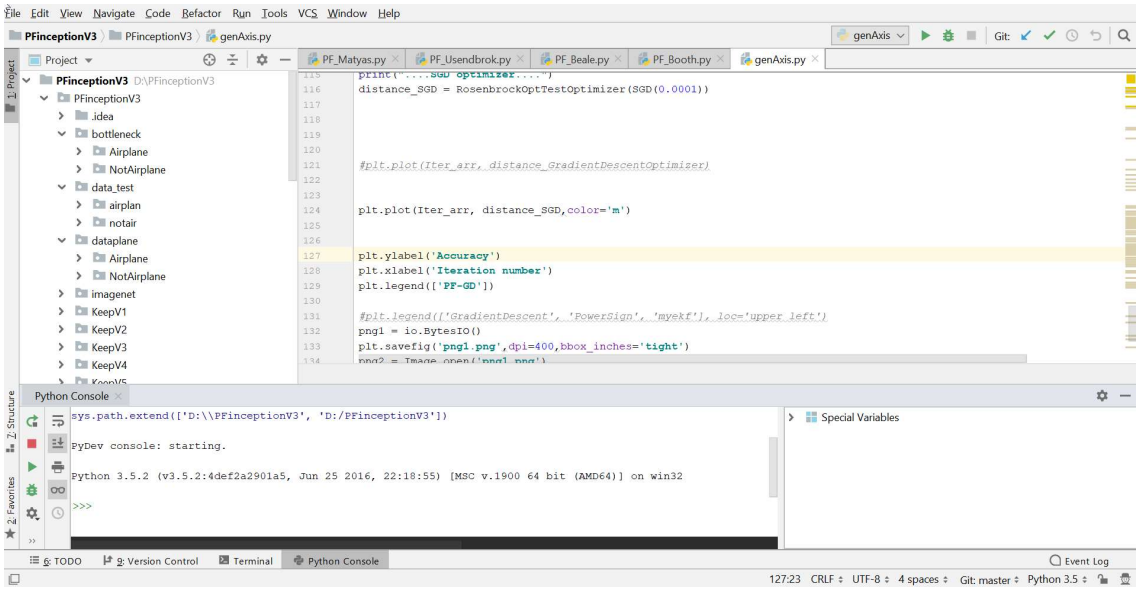
3.It supports a rich feature like cross-platform development so that the developers can write a script on different platforms as well.

4.PyCharm also comes with a good feature of the customizable interface which in turn increases the productivity.

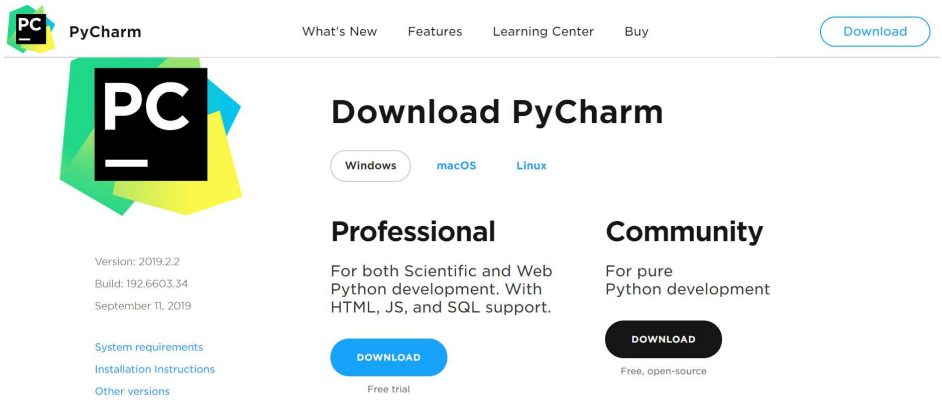
Cons:

1.PyCharm is an expensive tool while considering the features and the tools it provides to the client.

2.The initial installation is difficult and may hang up in between sometimes.



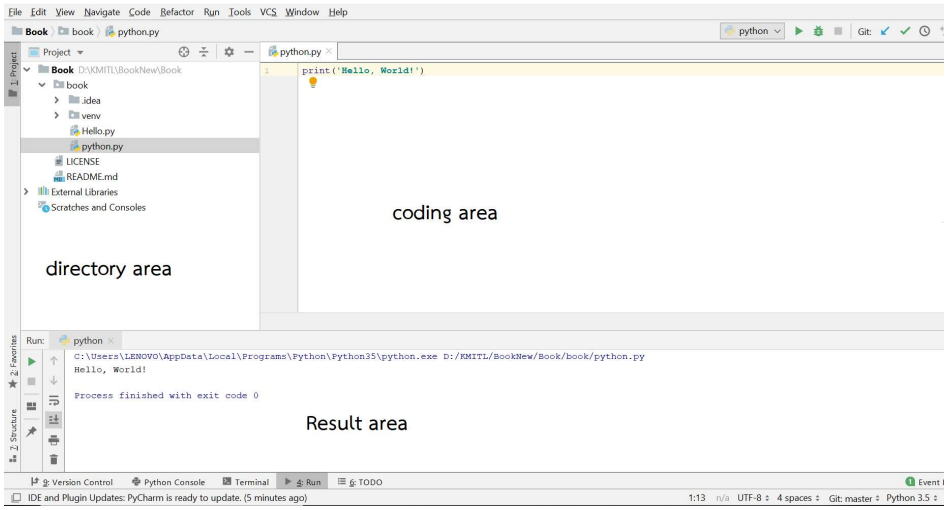
The official website for downloading PyCharm is https://www.jetbrains.com/pycharm/ . There are many version which suitable for different type of users [3].



For beginner, the Community version is enough for learning Python Programming. The next section is about Python executes or complier.

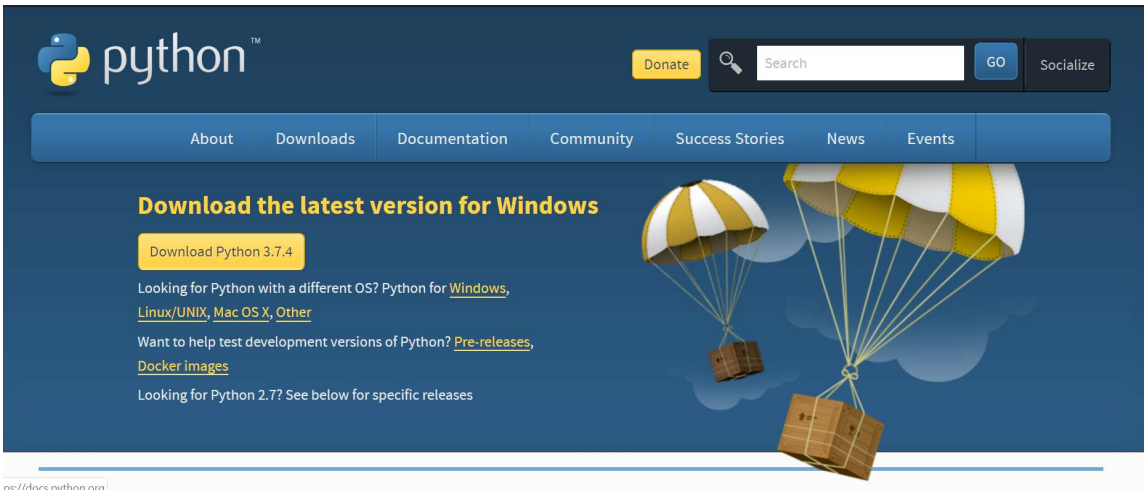
PyCharm Introduction

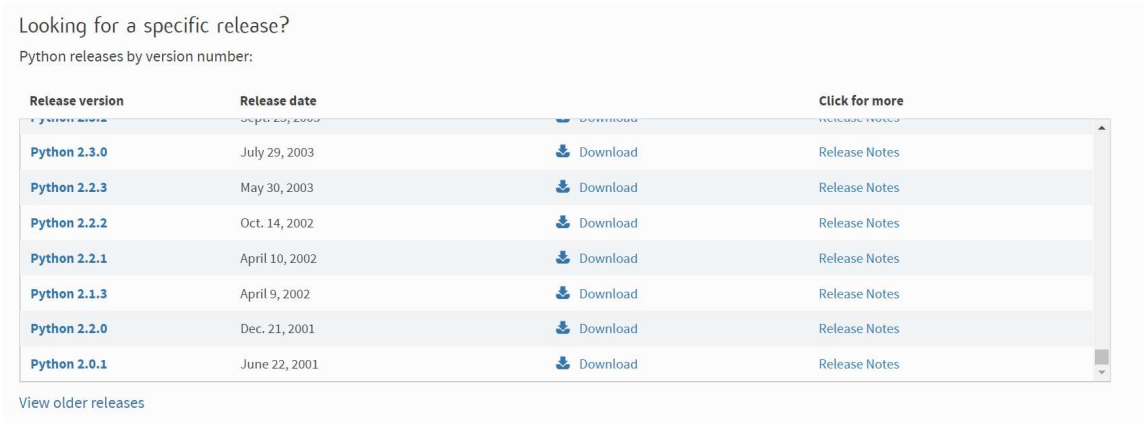
PyCharm have three mainly areas namely directory area, coding area, and result area. Directory area will show the directory or folder that you want to see. The coding area is a area that you can coding with Python syntax. The result area can demonstrate the result from running program or debug program.



1.1.2 Python executes(compiler)

Same as other computer programming, Python have various version and different version have different feature and some time have different spalling of syntax. For all computer programming, Programmers must learn and remember syntax to increase coding performance [4]. Python 2 and 3 are widely use depend on different application. For example, in deep learning application (a kind of artificial intelligent, AI) mostly implement with Python 3 because a supporting-libraries.





Looking for a specific release?

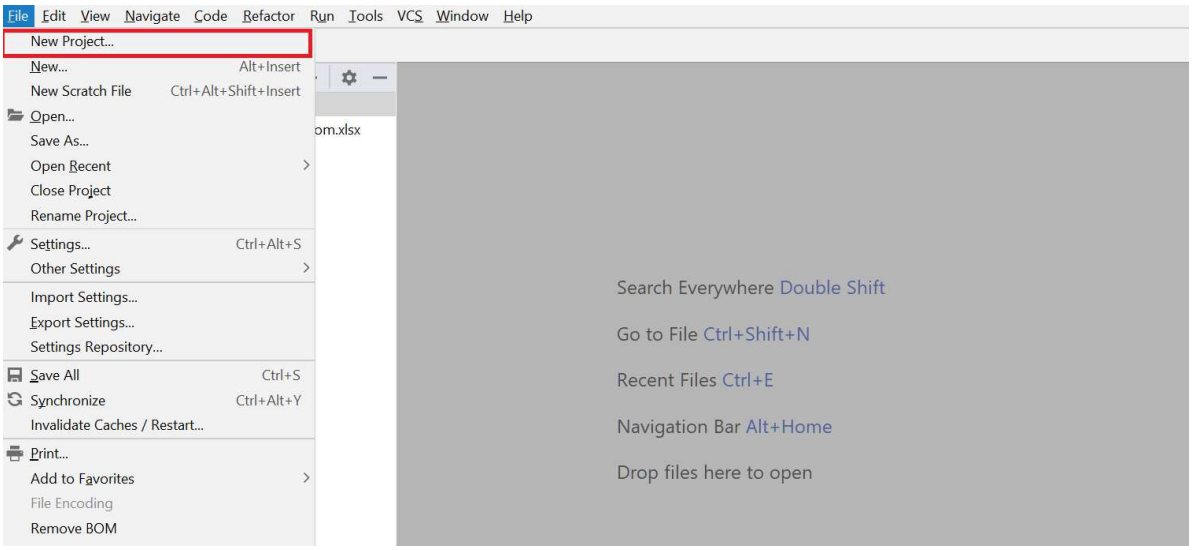
Python releases by version number:

Release version	Release date		Click for more
Python 2.3.0	July 29, 2003	Download	Release Notes
Python 2.2.3	May 30, 2003	Download	Release Notes
Python 2.2.2	Oct. 14, 2002	Download	Release Notes
Python 2.2.1	April 10, 2002	Download	Release Notes
Python 2.1.3	April 9, 2002	Download	Release Notes
Python 2.2.0	Dec. 21, 2001	Download	Release Notes
Python 2.0.1	June 22, 2001	Download	Release Notes

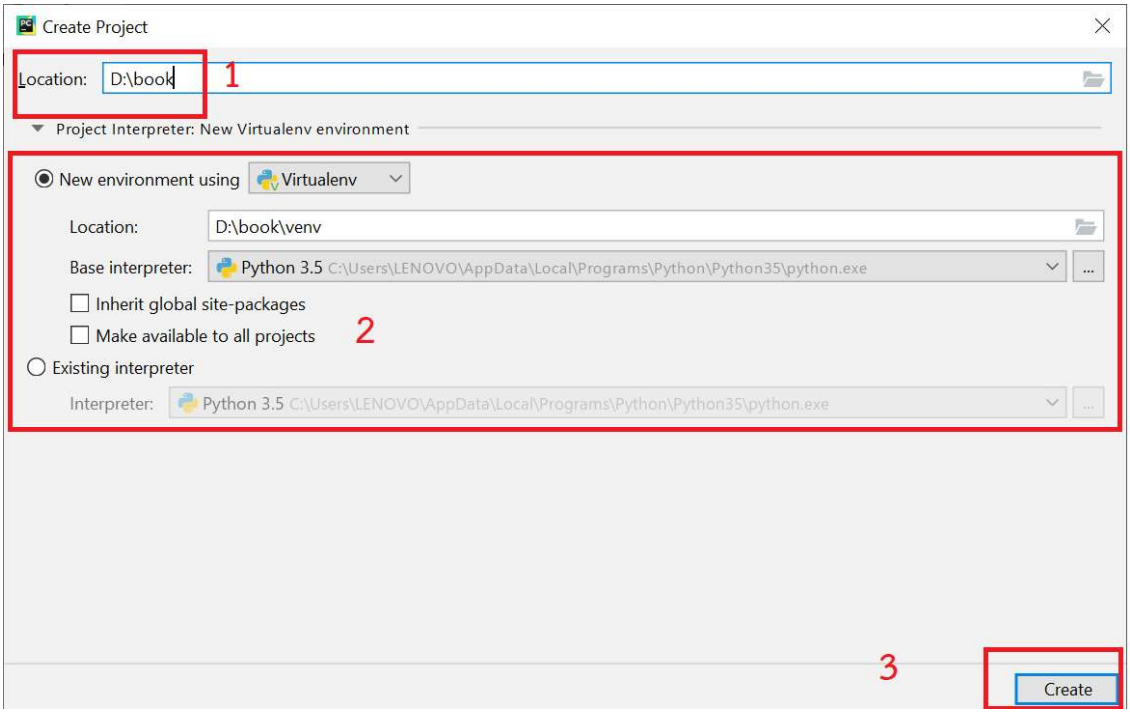
View older releases

1.2 The first program

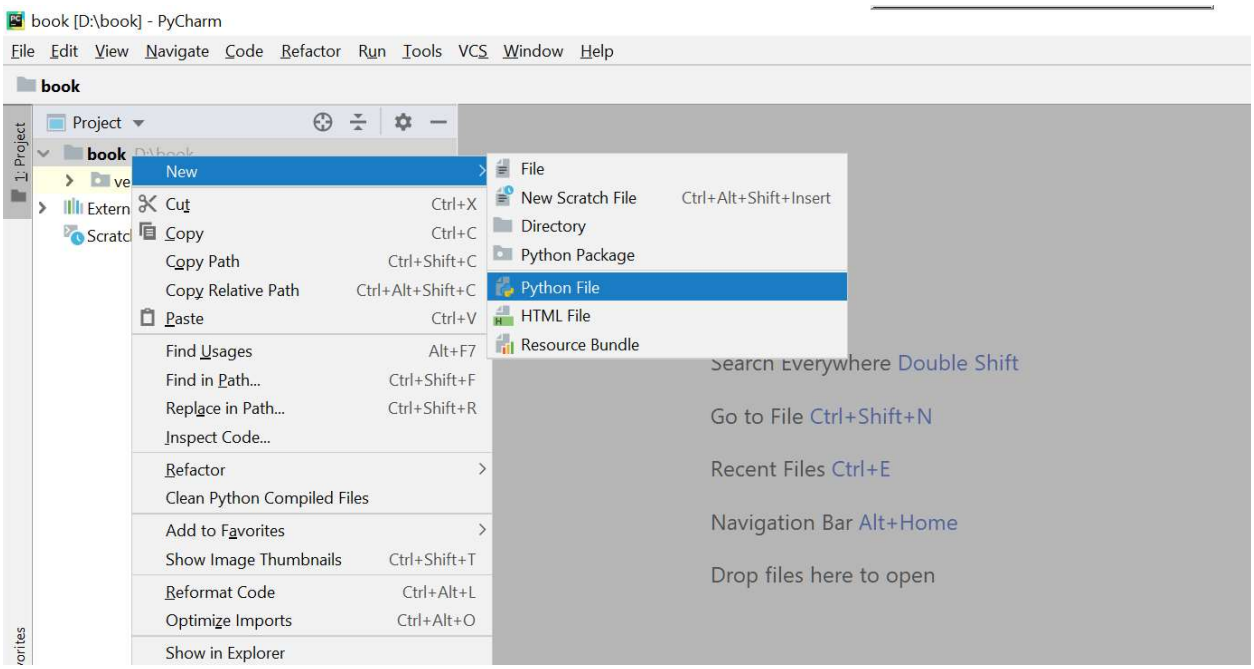
After install Python IDE and Python compiler, try to print sentence “Hello World” as the following code and suggestion. New Project as the below picture:



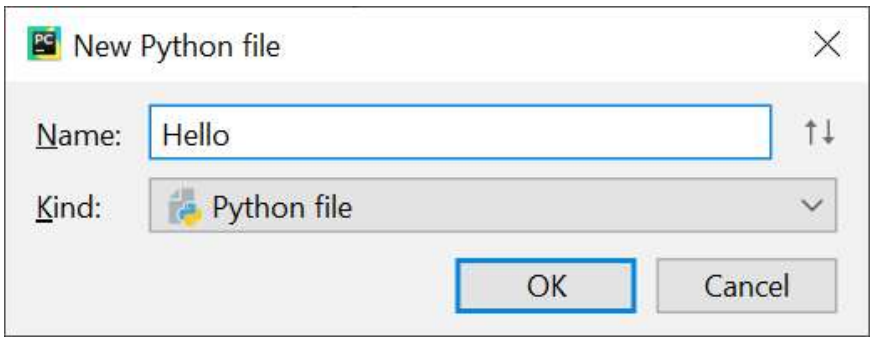
Set location for the new project, then select an interpreter by choosing the new environment or existing interpreter and create in the last step.



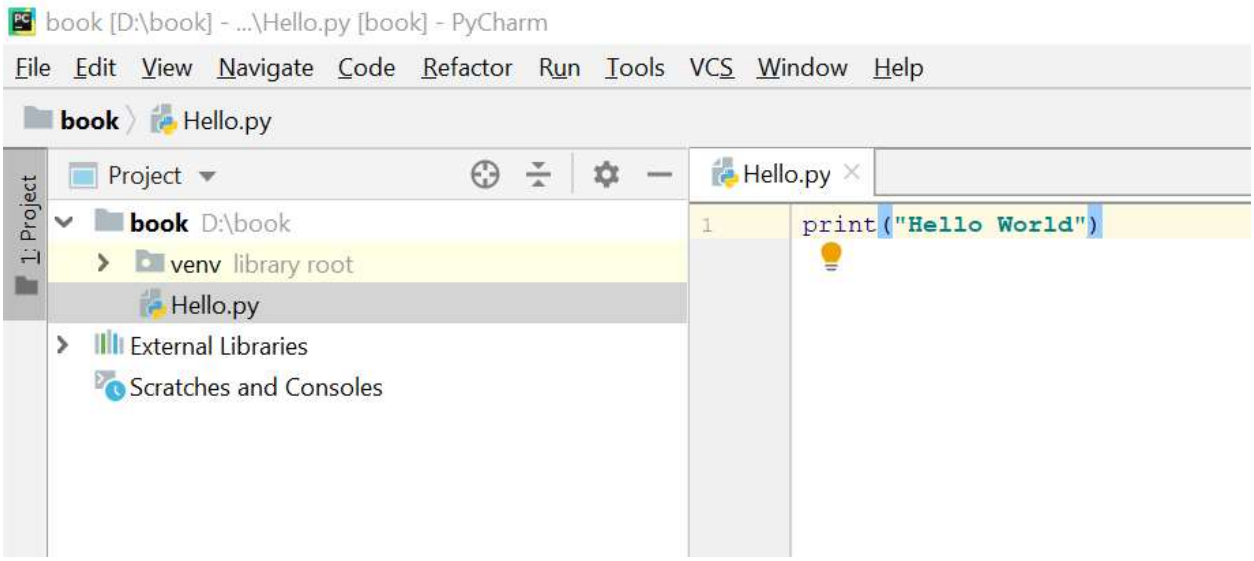
New Python File as the follow:



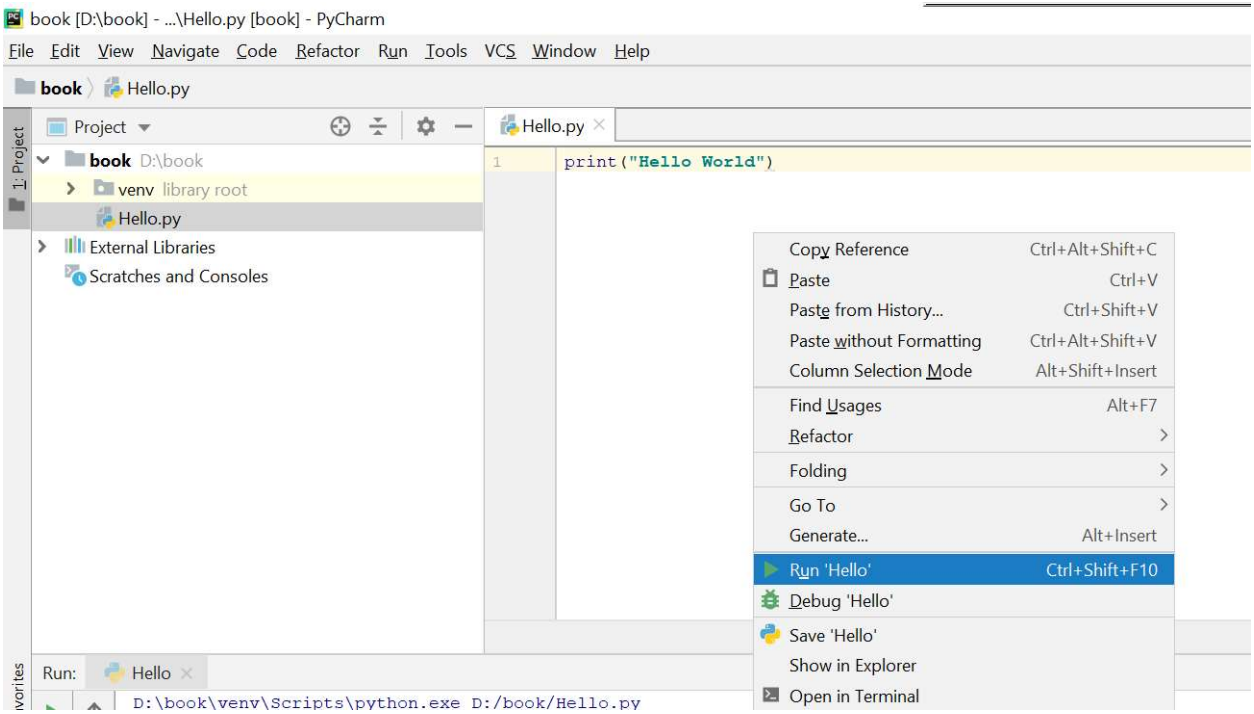
Type file name and then hit “OK”

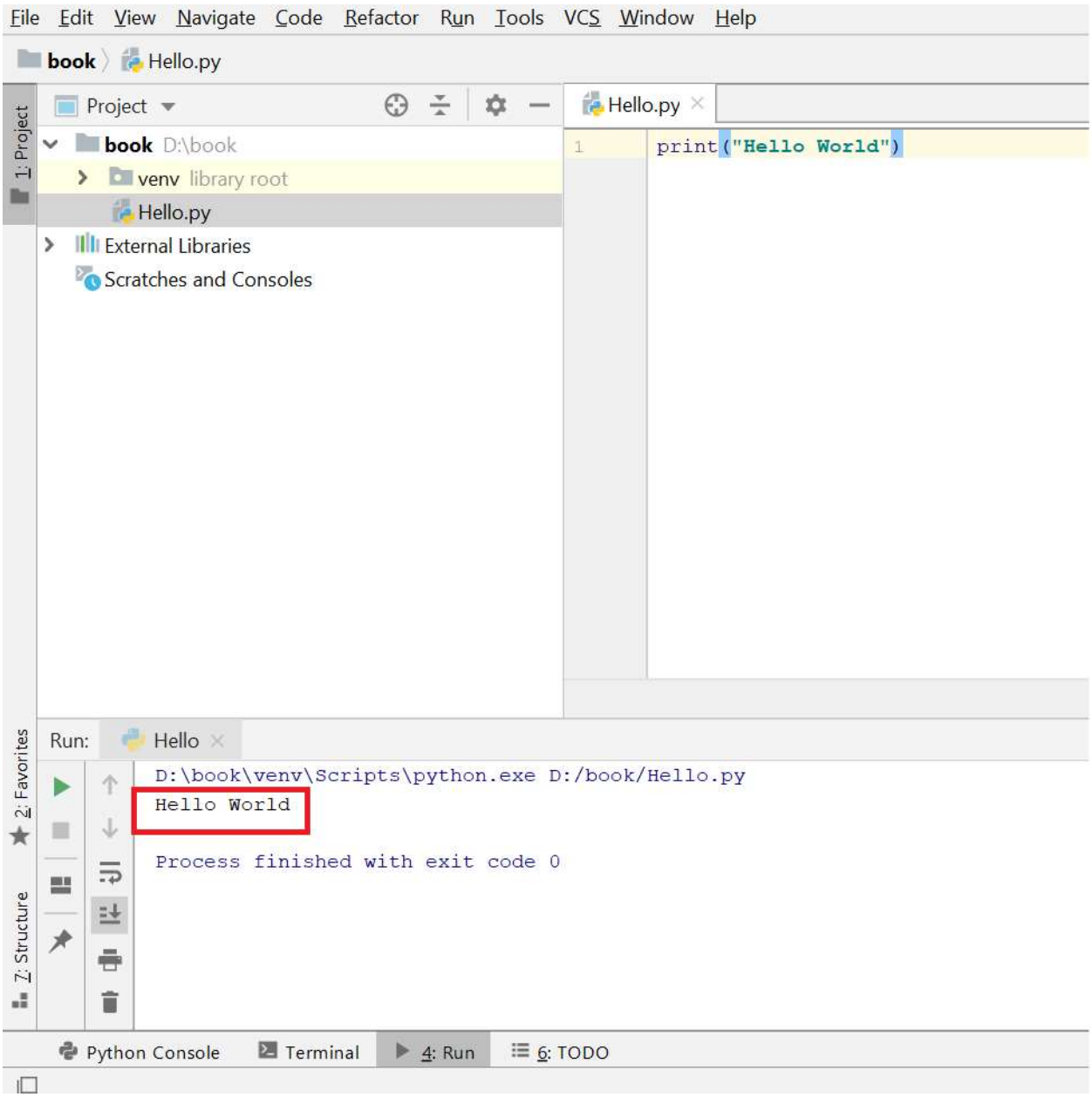


Type the first syntax => print(...) as following image



Right click on screen and select Run "youfilename"

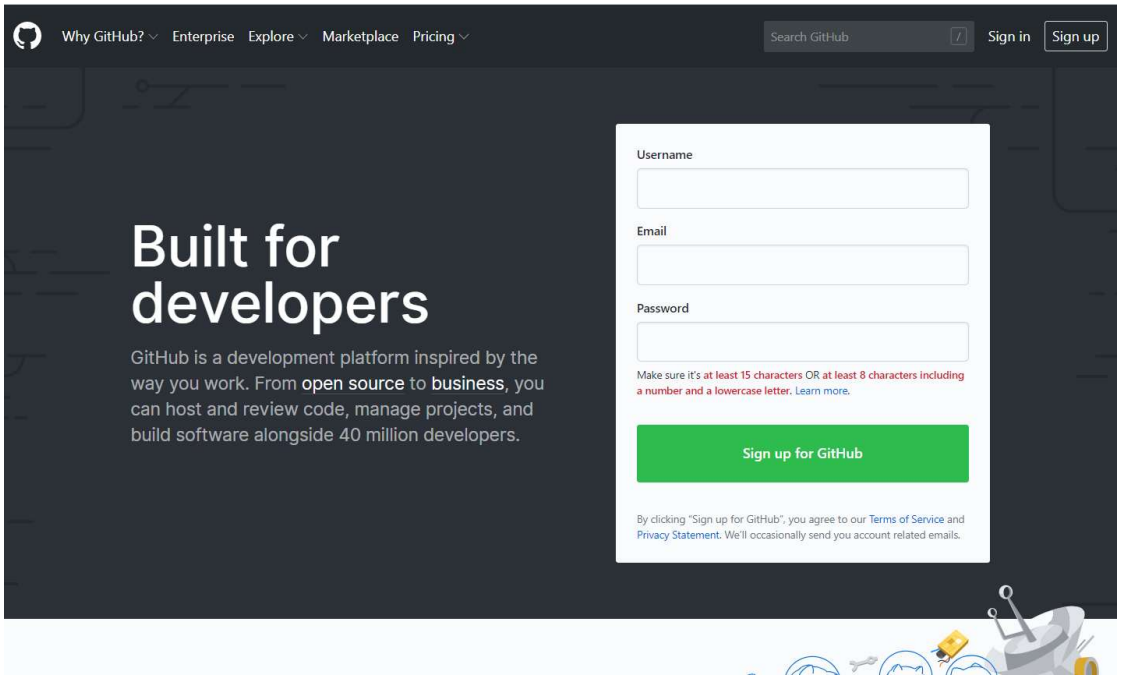


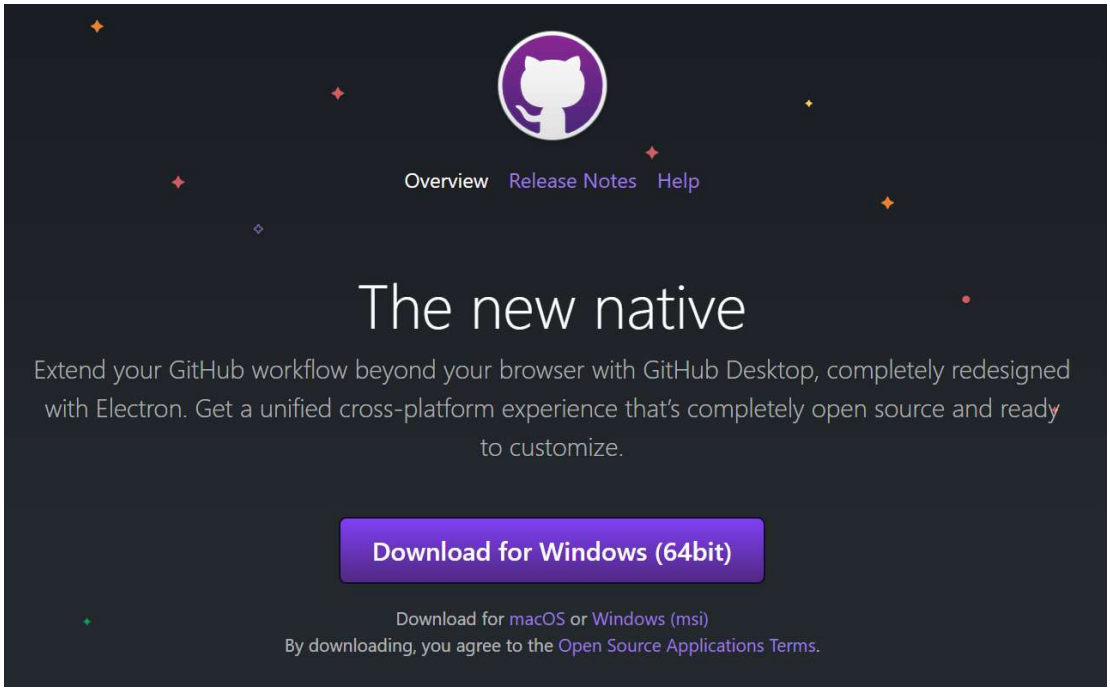


The result will show as follow with the word “Hello World”. For more information and syntax for each operation of Python is available in many website including [5].

1.3 Installing and Using GitHub

Now we finish the first program and the next step is learning about how to present your work. Presentation through website is normally for today. To build a website, https://github.com/ and GitHub Desktop [6] are implemented.

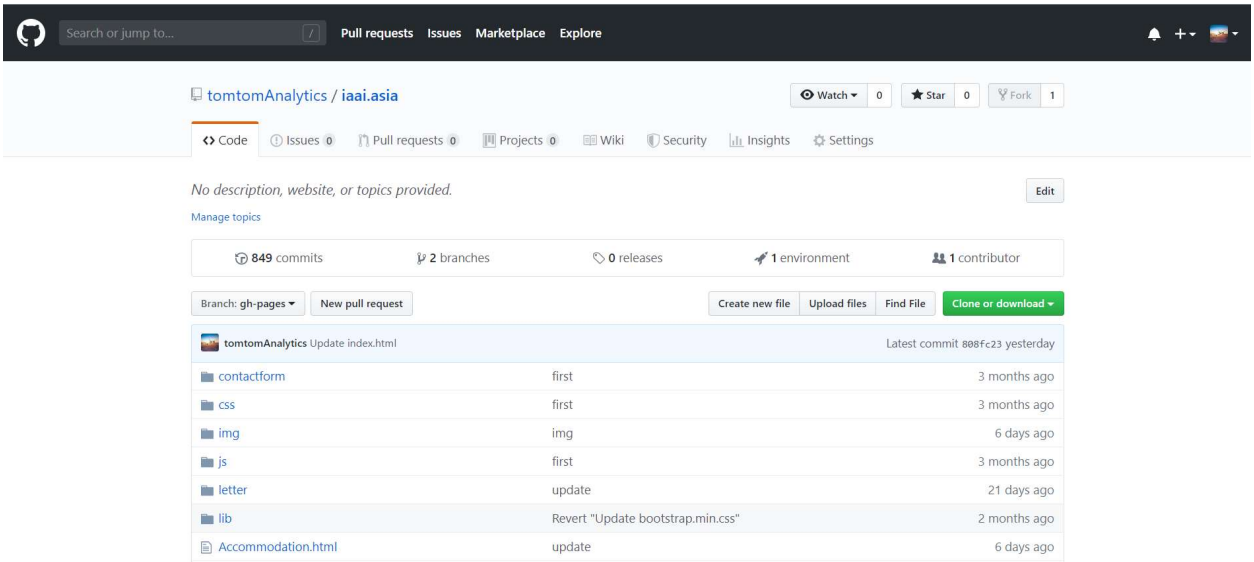




1.3.1 GitHub Introduction

GitHub is an American company that provides hosting for software development version control using Git. It is a subsidiary of Microsoft, which acquired the company in 2018 for $7.5 billion. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project [7].

GitHub offers plans for free, professional, and enterprise accounts.Free GitHub accounts are commonly used to host open source projects. As of January 2019, GitHub offers unlimited private repositories to all plans, including free accounts. As of May 2019, GitHub reports having over 37 million users and more than 100 million repositories (including at least 28 million public repositories), making it the largest host of source code in the world.



1.3.2 Version Control Using GitHub Desktop

What is Version Control and Why Use It?

It is helpful to understand what version control is and why it might be useful for the work you are doing prior to getting stuck into the practicalities. At a basic level version control involves taking 'snapshots' of files at different stages. Many people will have introduced some sort of version control systems for files. Often this is done by saving different versions of the files. Something like this:

```
mydocument.txt
mydocumentversion2.txt
mydocumentwithrevision.txt
mydocumentfinal.txt
```

The system used for naming files may be more or less systematic. Adding dates makes it slightly easier to follow when changes were made:

```
mydocument2016-01-06.txt
mydocument2016-01-08.txt
```

Though this system might be slightly easier to follow, there are still problems with it. Primarily this system doesn't record or describe the changes that took place between these two saves. It is possible that some of these changes were small typo fixes but the changes could also have been a major re-write or re-structuring of a document. If you have a change of heart about

some of these changes you also need to work out which date the changes were made in order to go back to a previous version.

Version control tries to address problems like these by implementing a systematic approach to recording and managing changes in files. At its simplest, version control involves taking 'snapshots' of your file at different stages. This snapshot records information about when the snapshot was made but also about what changes occurred between different snapshots. This allows you to 'rewind' your file to an older version. From this basic aim of version control a range of other possibilities are made available.

What are Git and GitHub?

Though often used synonymously, Git and GitHub are two different things. Git is a particular implementation of version control originally designed by Linus Torvalds as a way of managing the Linux source code. Other systems of version control exist though they are used less frequently. Git can be used to refer both to a particular approach taken to version control and the software underlying it.

GitHub is a company which hosts Git repositories (more on this below) and provides software for using Git. This includes 'GitHub Desktop' which will be covered in this tutorial. GitHub is currently the most popular host of open source projects by number of projects and number of users.

Although GitHub's focus is primarily on source code, other projects, such as the Programming Historian, are increasingly making use of version control systems like GitHub to

manage the work-flows of journal publishing, open textbooks and other humanities projects. Becoming familiar with GitHub will be useful not only for version controlling your own documents but will also make it easier to contribute and draw upon other projects which use GitHub. In this lesson the focus will be on gaining an understanding of the basic aims and principles of version control by uploading and version controlling a plain text document. This lesson will not cover everything but will provide a starting point to using version control.

Why Not use Dropbox or Google Drive?

Dropbox, Google Drive and other services offer some form of version control in their systems. There are times when this may be sufficient for your needs. However, there are a number of advantages to using a version control system like Git:

- Language support: Git supports both text and programming languages. As research moves to include more digital techniques and tools it becomes increasingly important to have a way of managing and sharing both the 'traditional' outputs (journal articles, books, etc.) but also these newer outputs (code, datasets etc.)
- More control: a proper version control systems gives you a much greater deal of control over how you manage changes in a document.
- Useful history: using version control systems like Git will allow you to produce a history of your document in which different stages of the documents can be navigated easily both by yourself and by others.

1.3.3 Using GitHub and GitHub Desktop

Getting Started

GitHub Desktop will allow us to easily start using version control. GitHub Desktop offers a Graphical User Interface (GUI) to use Git. A GUI allows users to interact with a program using a visual interface rather than relying on text commands. Though there are some potential advantages to using the command line version of Git in the long run, using a GUI can reduce the learning curve of using version control and Git. If you decide you are interested in using the command line you can find more resources at the end of the lesson.

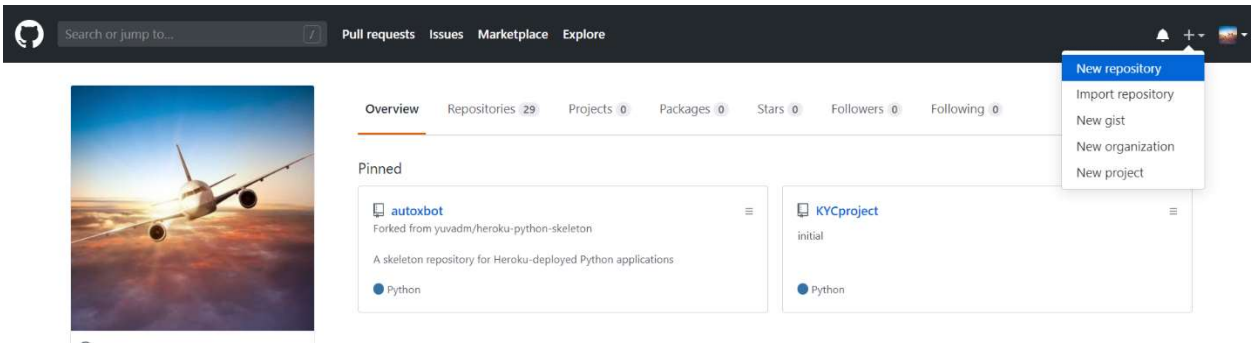
Register for a GitHub Account

Since we are going to be using GitHub we will need to register for an account at GitHub if we don't already have one. For students and researchers GitHub offers free private repositories. These are not necessary but might be appealing if you want to keep some work private.

Install GitHub Desktop

Once you have downloaded the file, unzip it and open the app, following the instructions for logging in to your GitHub account. Once you have installed GitHub Desktop and followed the setup instructions, we can start using the software with a text document.

Creating Repository in GitHub



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner tomtomAnalytics / **Repository name** * Book

Great repository names are short and memorable. Need inspiration? How about **shiny-umbrella**?

Description (optional)

Public
Anyone can see this repository. You choose who can commit.

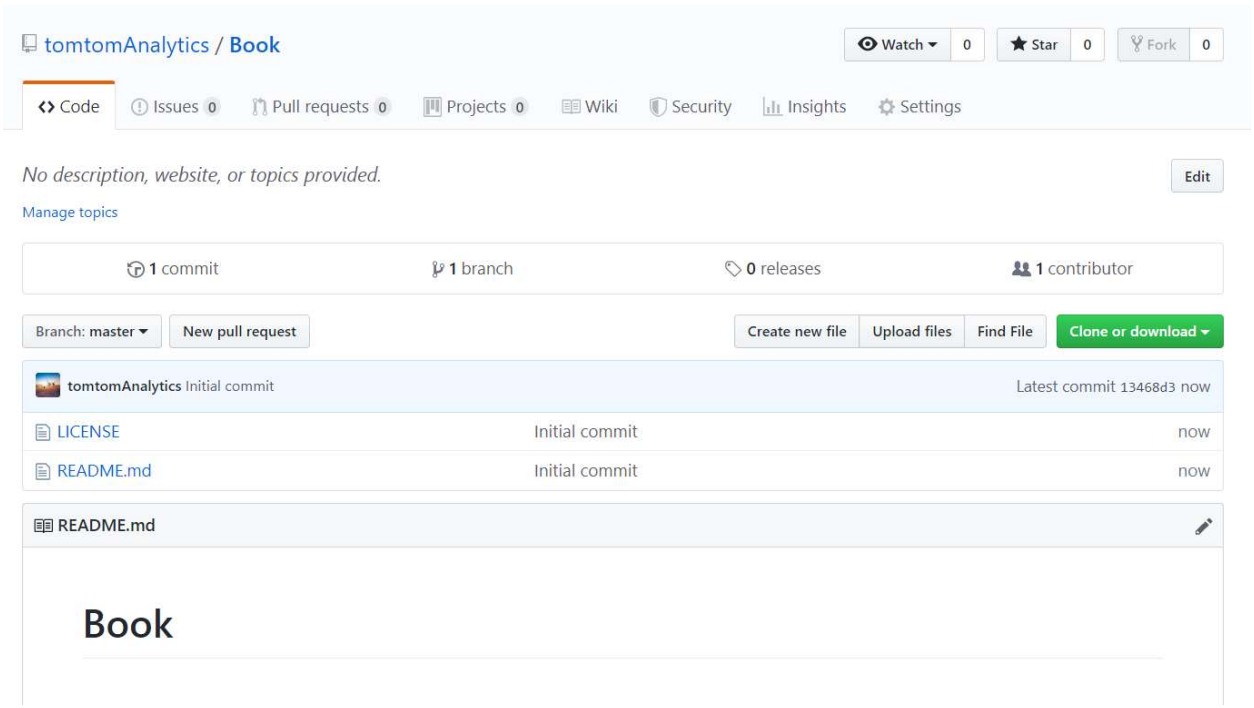
Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

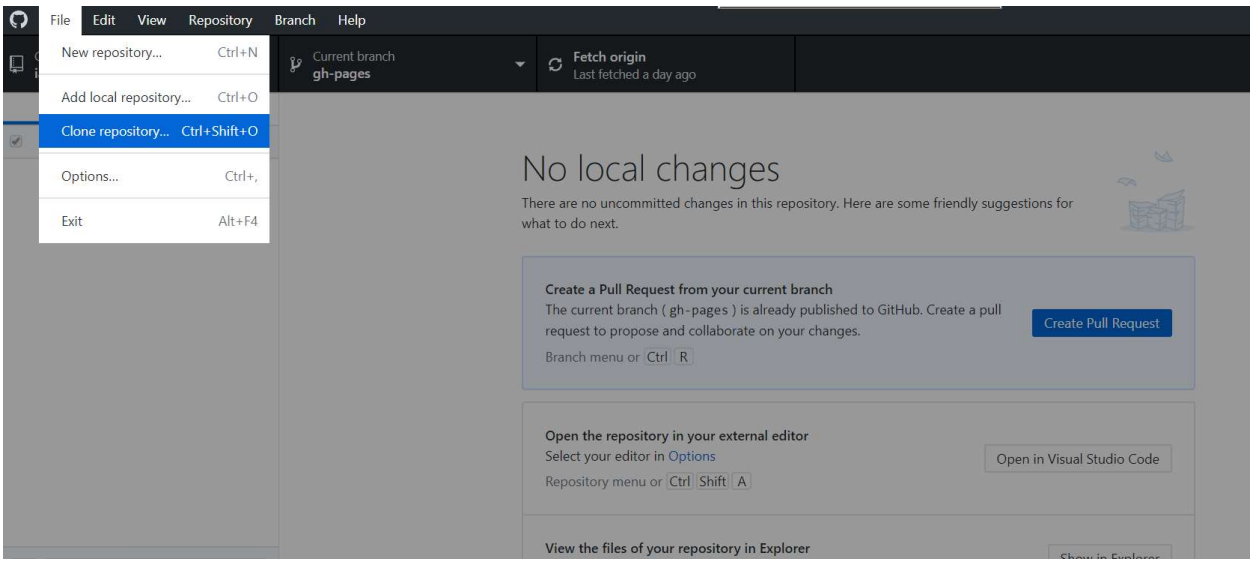
Initialize this repository with a README
This will let you immediately clone the repository to your computer.

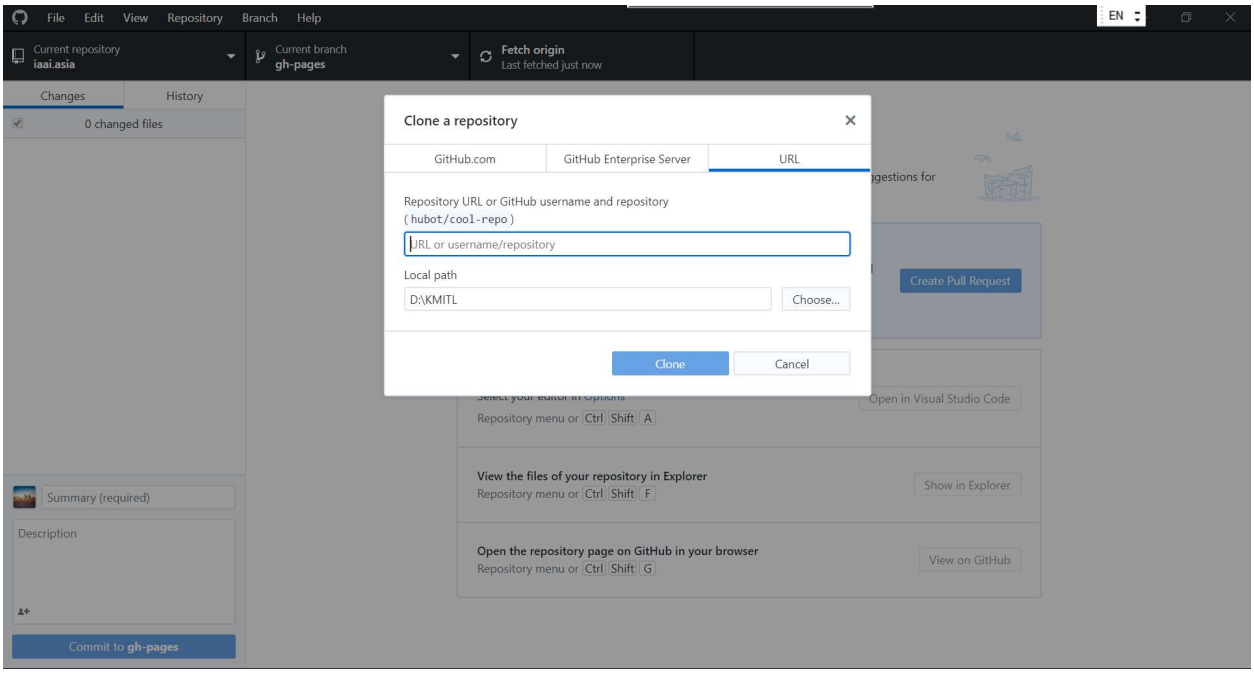
Add .gitignore: None | Add a license: GNU General Public License v3.0

Create repository

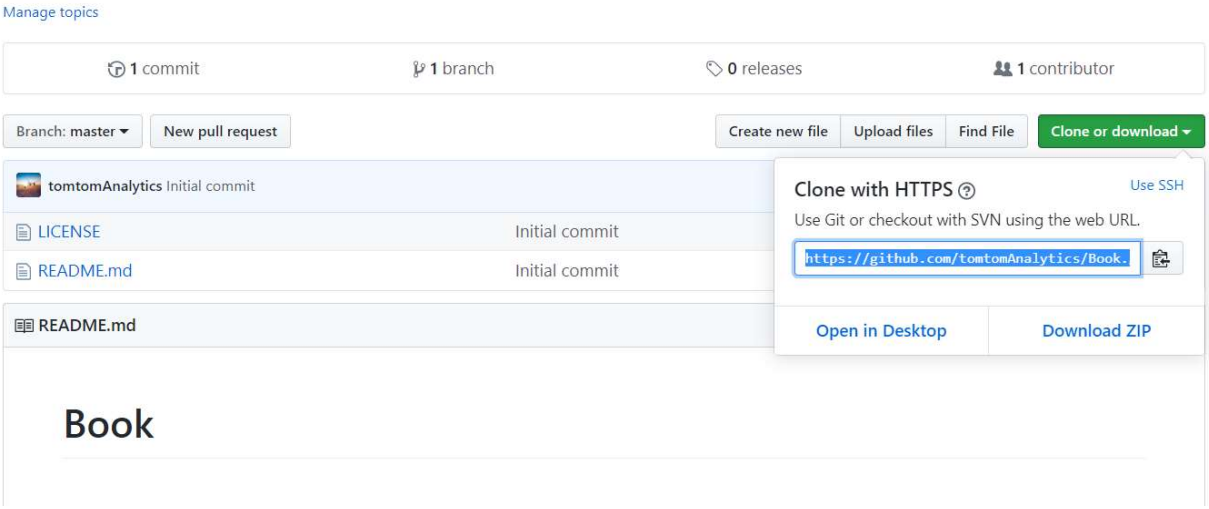


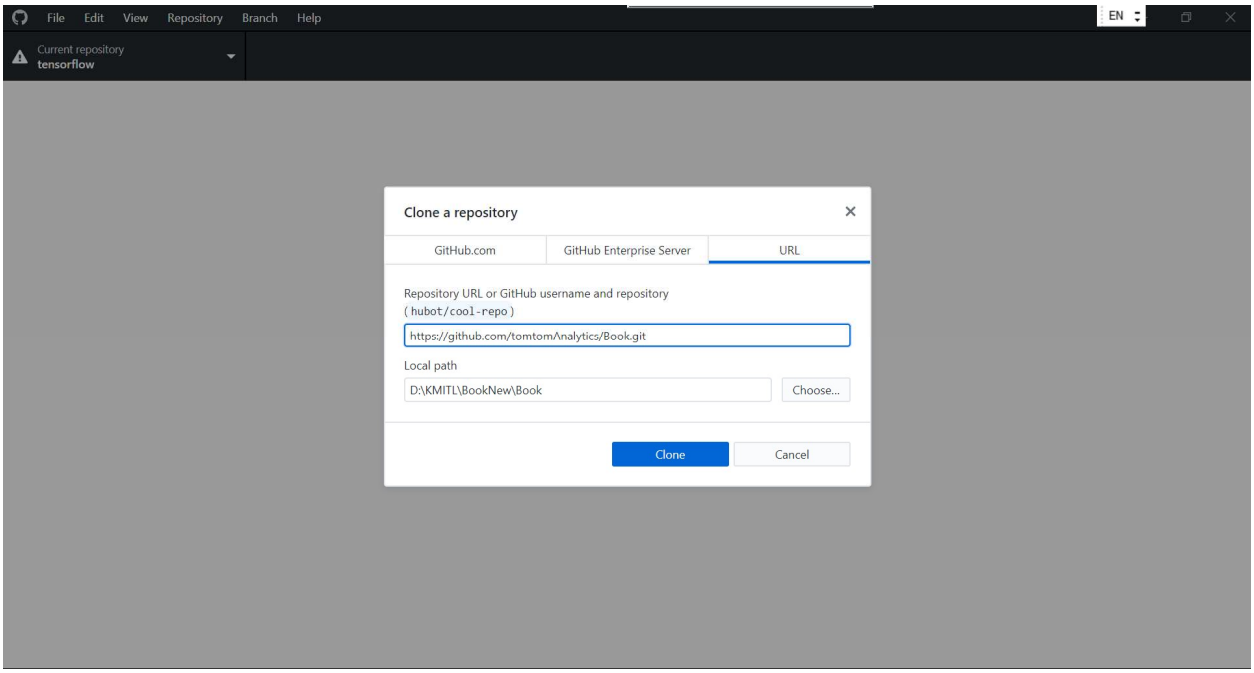
Clone Repository to Local by GitHub Desktop



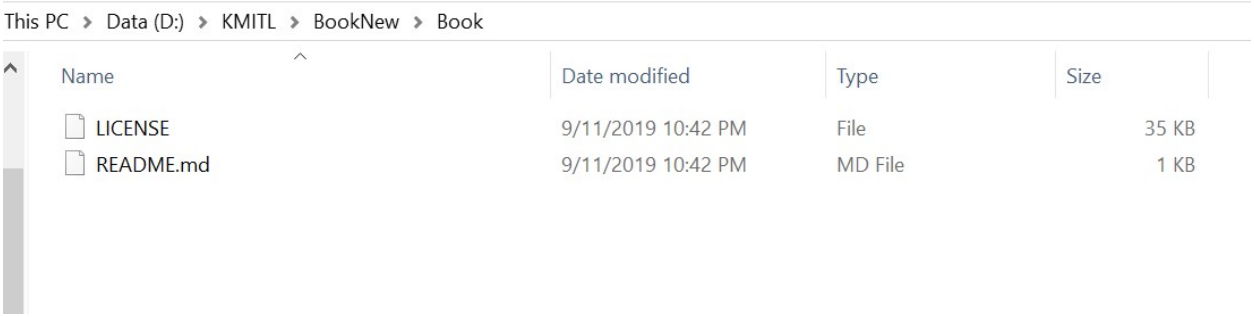


Copy HTTPS path from GitHub account and paste to GitHub Desktop, then select the location in your computer and following with Clone in the last step.





After clone Repository process finish, the result will show as the following figure.

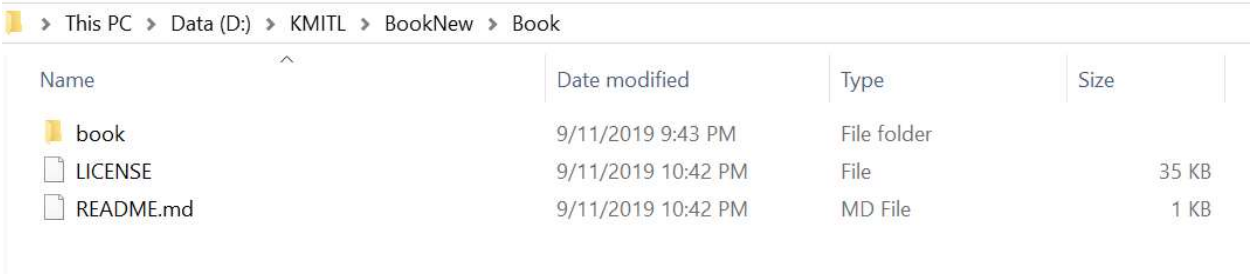


Creating a Document

We can begin with a very simple document such as Hello.py as the previous section.

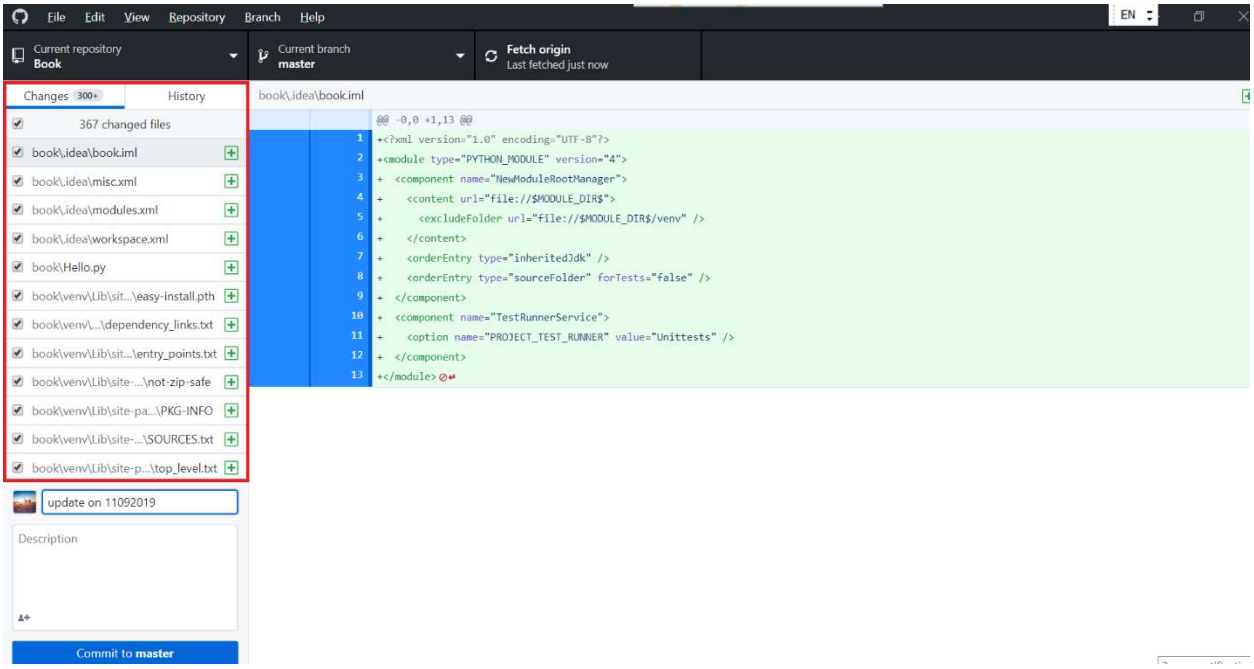
To most effectively use Git to version control it is important to organize projects in folders. Git tracks the contents of a folder by creating a repository in the folder. The repository is made up of all the files in the folder that are 'watched' for changes by Git. It is best to create one repository for each major project you are working on, i.e., one repository for an article, one for a book, and one for some code you are developing. These folders are like the normal folders you

would have on your computer for different projects, though the files in the folders have to be deliberately added to the repository in order to be version controlled.



Adding a Document

There are a number of different ways to add files for GitHub Desktop to track. We can drag the folder containing the file onto GitHub Desktop.

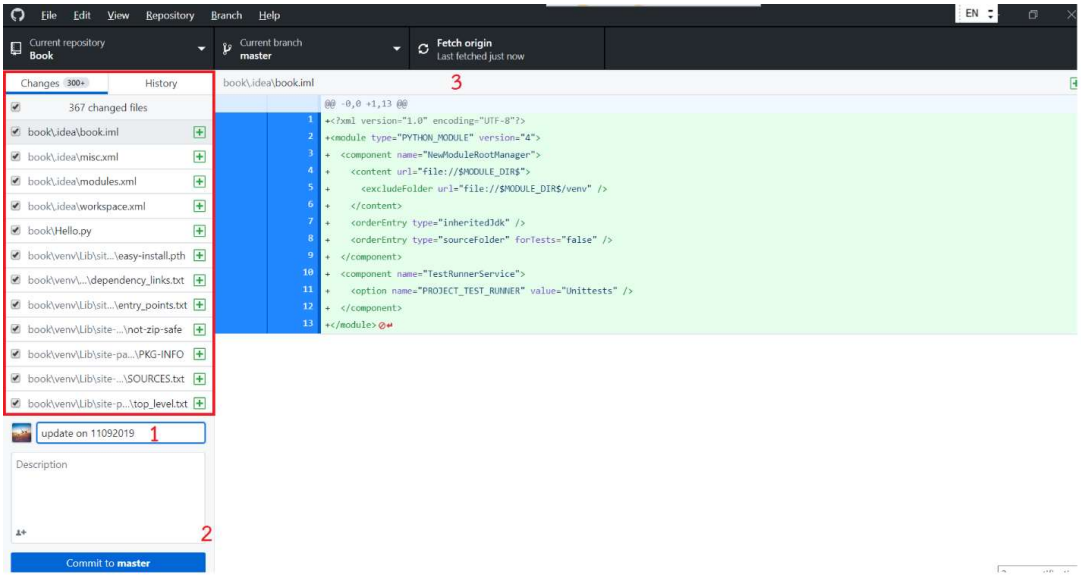


Committing Changes

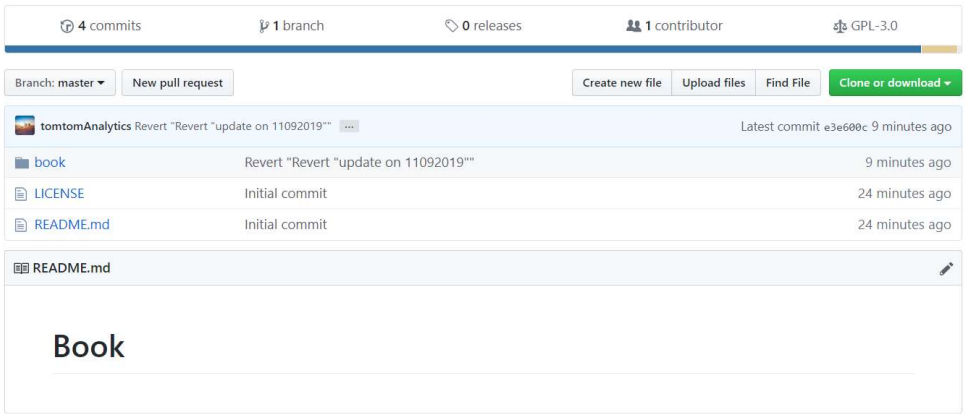
A commit tells Git that you made some changes which you want to record. Though a commit seems similar to saving a file, there are different aims behind 'committing' changes

compared to saving changes. Though people sometimes save different versions of a document, often you are saving a document merely to record the version as it is when it is saved. Saving the document means you can close the file and return to it in the same state later on. Commits, however, take a snapshot of the file at that point and allow you to document information about the changes made to the document.

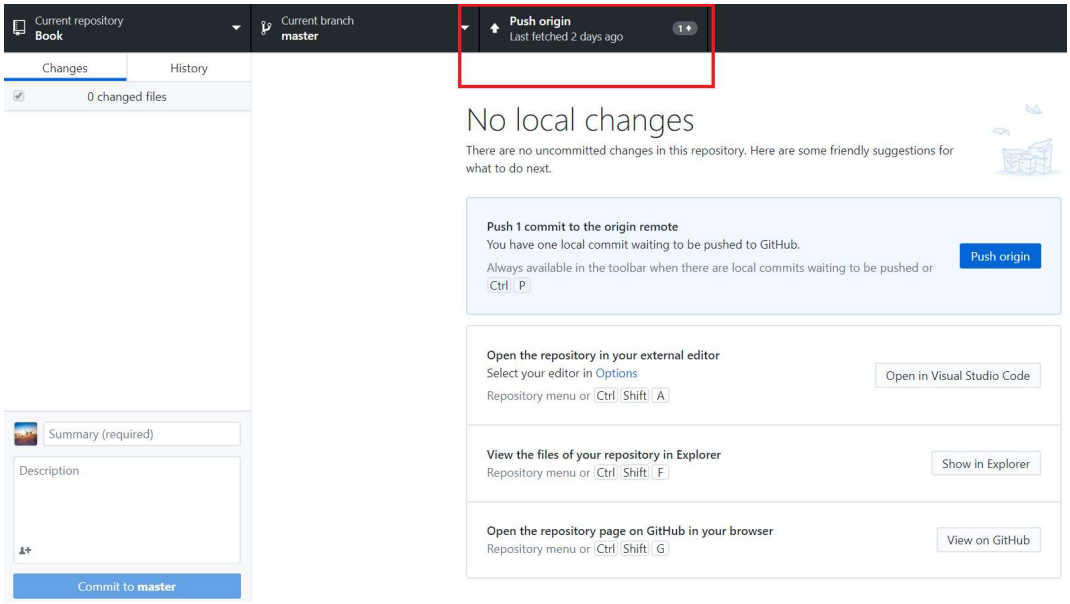
The method to commit a change is type some note in the box Summary(required) and press a Commit to master and following with Fetch origin to push all change to GitHub account.



Result



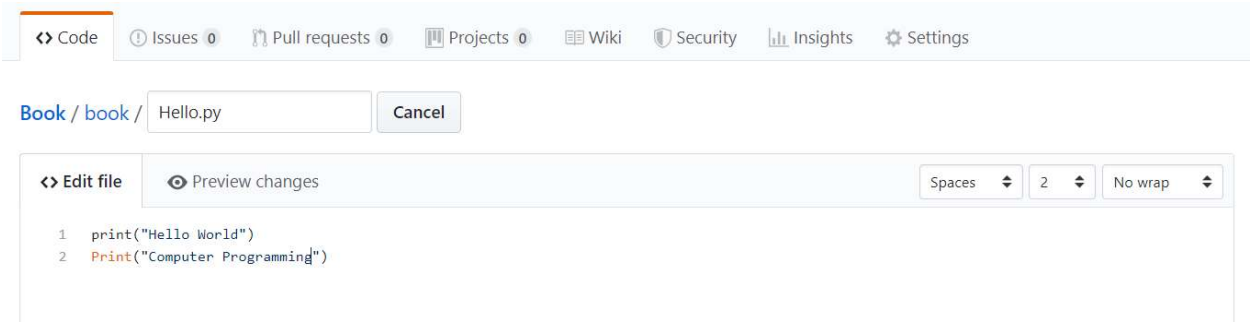
The next step is Push Origin to update the changing code in GitHub server.



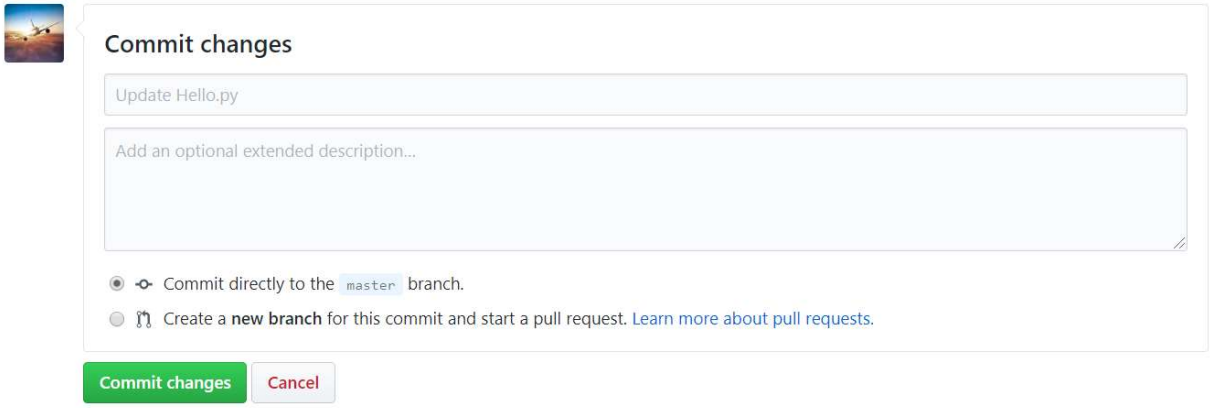
Making Changes Remotely

It is also possible to make a change to your repository on the web interface. Clicking on the name of the file will take you to a new page showing your document. From the web interface you have a variety of options available to you, including viewing the history of changes, viewing the file in GitHub Desktop, and deleting it. You can also see some other options next to 'code'. These options will not be so important to begin with but you may want to use them in the future. For now we will try editing a file in the web interface and syncing these changes to our local repository.

You will now be able to edit the file and add some new text.



Once you have made some changes to your file, you will again see the option to commit changes at the bottom of the text entry box.



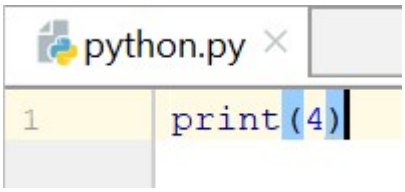
Once you have committed these changes they will be stored on the remote repository. To get them back onto our computer we need to sync our these changes. We will see the 'Push Origin' button on GitHub Desktop.

Chapter 2 Variables, expressions and statements

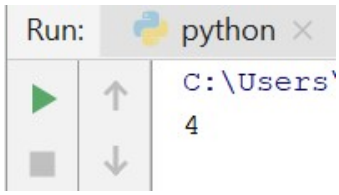
2.1 Values and types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'. These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. The print statement also works for integers.

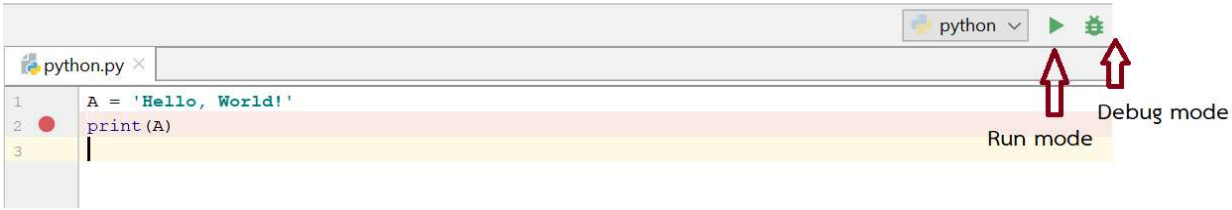
Code



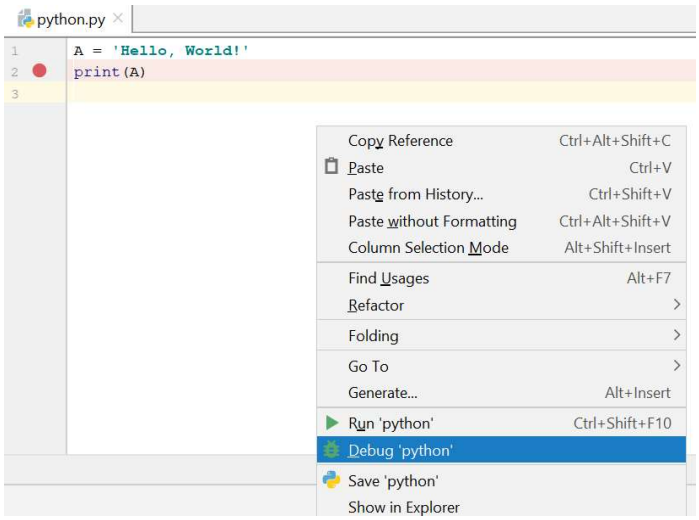
Result



If you are not sure what type a value has, the interpreter can tell you by setting debug line (result in red line) before selecting Debug mode in PyCharm

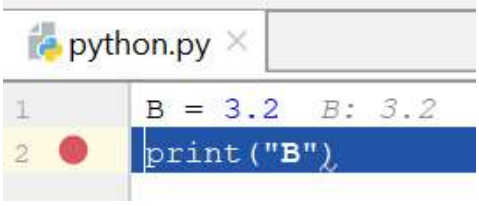


Or right click in coding area and then select debug mode. The result will show in result area as the following figure.

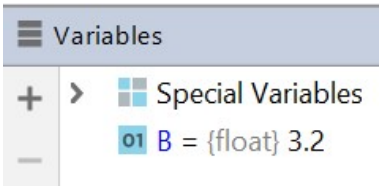


Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.

Code

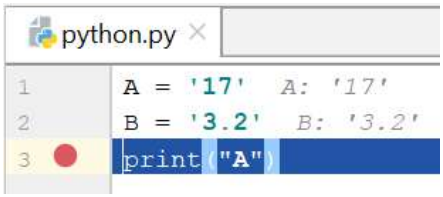


Result

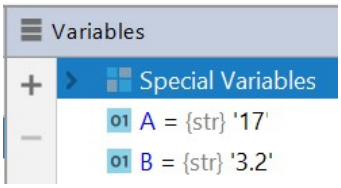


What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

Code

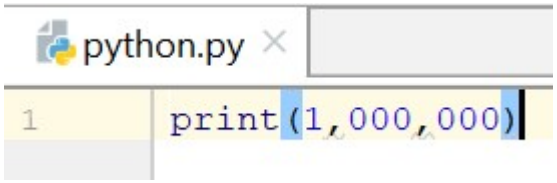


Result



They’re strings. When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

Code



Result

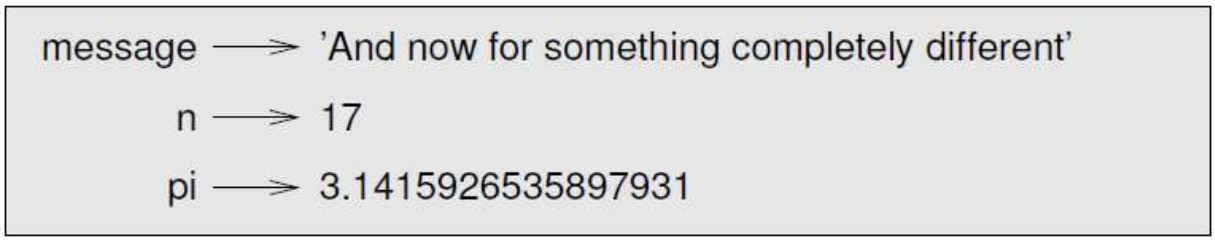
Well, that’s not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers, which it prints with spaces between. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn’t do the “right” thing.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value. An **assignment statement** creates new variables and gives them values:

```
python.py
1  message = 'And now for something completely different'
2  n = 17
3  pi = 3.14159265358979311
```

This example makes three assignments. The first assigns a string to a new variable named message; the second gives the integer 17 to n; the third assigns the (approximate) value of π to pi. A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the previous example:



To display the value of a variable, you can use a print statement:

Code

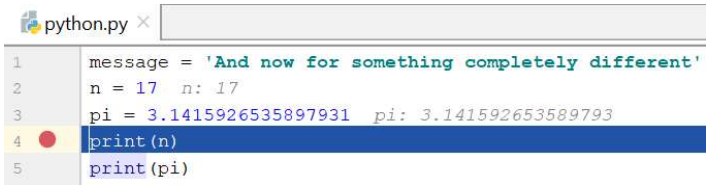
```
python.py
1  message = 'And now for something completely different'
2  n = 17
3  pi = 3.1415926535897931
4  print(n)
5  print(pi)
```

Result

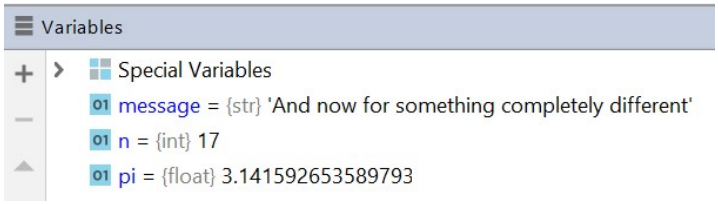
```
python
C:\Users\LENOVO\Ap
17
3.141592653589793
```

The type of a variable is the type of the value it refers to.

Code



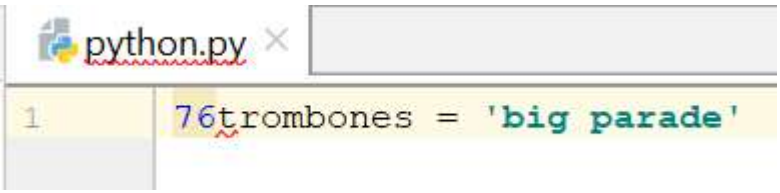
Result



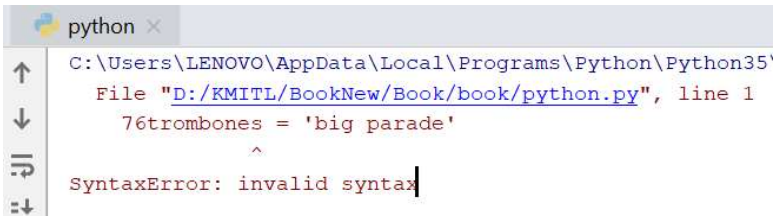
2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later). The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or airspeed_of_unladen_swallow. If you give a variable an illegal name, you get a syntax error:

Code



Result



76trombones is illegal because it does not begin with a letter.

It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names. Python has 31 keywords:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

2.4 Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment. When you type a statement in interactive mode, the

interpreter executes it and displays the result, if there is one.A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute. For example, the script

print(1)

x = 2

print(x)

produces the output

1

2

The assignment statement produces no output.

2.5 Operators and operands

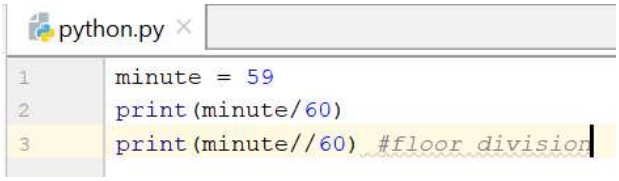
Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands. The operators +, -, *, / and ** perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

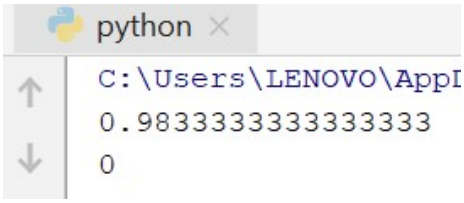
In some other languages, ^ is used for exponentiation, but in Python it is a bitwise operator called XOR. I won't cover bitwise operators in this book, but you can read about them at wiki.python.org/moin/BitwiseOperators.

The division operator might not do what you expect:

Code



Result



The value of minute is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **floor division**. When both of the operands are integers, the result is also an integer; floor division chops off the fraction part, so in this example it rounds down to zero. If either of the operands is a floating-point number, Python performs floating-point division, and the result is a float:

2.6 Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

17

x

x + 17

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
- Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2p, you can use parentheses or write degrees / 2 / pi.

2.8 String operations

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

'2'-'1'

'eggs'/'easy'

'third'*'a charm'

The + operator works with strings, but it might not do what you expect: it performs concatenation, which means joining the strings by linking them end-to-end. For example:

first = 'throat'

second = 'warbler'

print first + second

The output of this program is throatwarbler.

The * operator also works on strings; it performs repetition. For example, 'Spam'*3 is 'SpamSpamSpam'. If one of the operands is a string, the other has to be an integer. This use of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect 'Spam'*3 to be the same as 'Spam'+'Spam'+'Spam', and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

2.9 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the program. Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is much more useful to explain why. This comment is redundant with the code and useless:

```
v = 5 # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5 # velocity in meters/second.
```

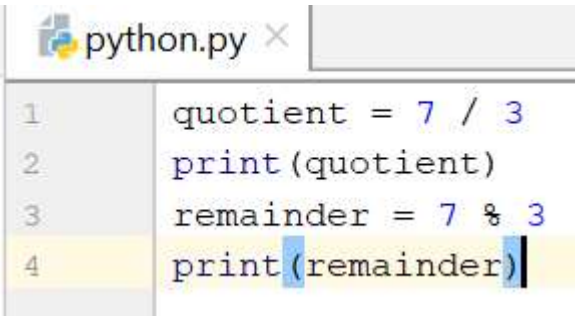
Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

Chapter 3 Conditionals

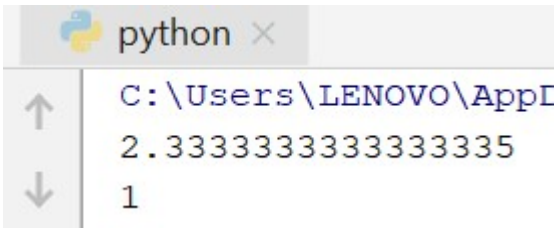
3.1 Modulus operator

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

Code



Result



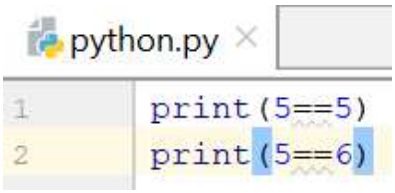
So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y. Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

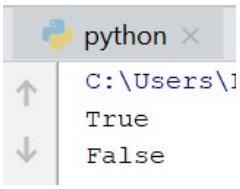
3.2 Boolean expressions

A boolean expression is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

Code



Result



The == operator is one of the comparison operators; the others are:

x != y # x is not equal to y

x > y # x is greater than y

x < y # x is less than y

x >= y # x is greater than or equal to y

x <= y # x is less than or equal to y

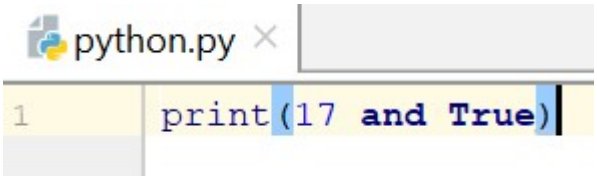
Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

3.3 Logical operators

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is

greater than 0 and less than 10. n%2 == 0 or n%3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3. Finally, the *not* operator negates a Boolean expression, so *not* (x > y) is true if x > y is false, that is, if x is less than or equal to y. Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

Code



Result

3.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
if x > 0:
        print 'x is positive'
```

The boolean expression after the if statement is called the condition. If it is true, then the indented statement gets executed. If not, nothing happens. if statements have the same structure as function definitions: a header followed by an indented block. Statements like this are called compound statements. There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements

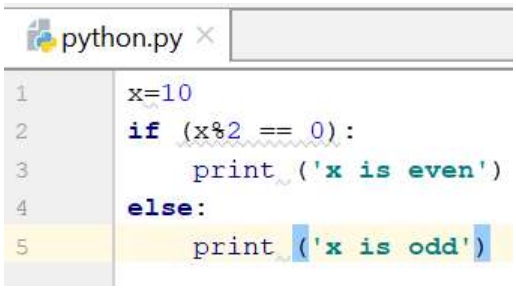
(usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0:
    pass # need to handle negative values!
```

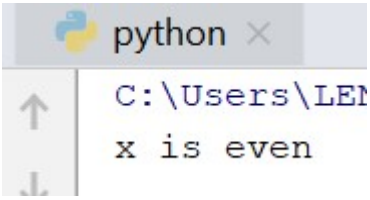
3.5 Alternative execution

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

Code



Result



If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

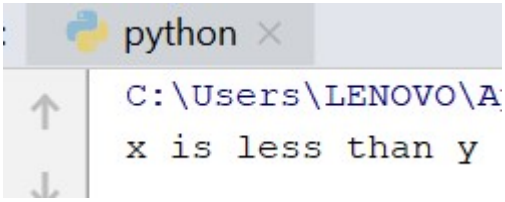
3.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

Code

```
x = 1
y = 5
if (x < y):
    print ('x is less than y')
elif (x > y):
    print ('x is greater than y')
else:
    print ('x and y are equal')
```

Result



elif is an abbreviation of “else if.” Again, exactly one branch will be executed. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

```
x = 1
y = 5
if (x < y):
    print ('x is less than y')
elif (x > y):
    print ('x is greater than y')
elif (x == y):
    print ('x and y are equal')
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

3.7 Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example like this:

python.py

```
x = 1
y = 5
if (x == y):
    print ('x and y are equal')
else:
    if (x < y):
        print ('x is less than y')
    else:
        print ('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

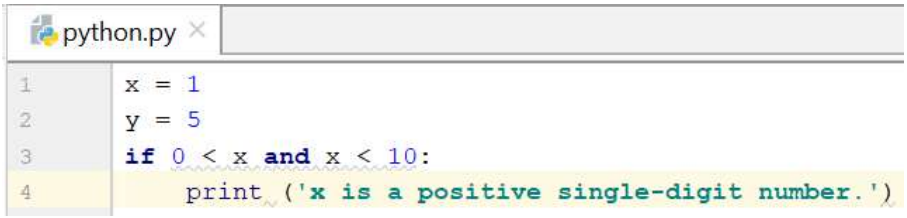
Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

python.py

```
x = 1
y = 5
if 0 < x:
    if x < 10:
        print ('x is a positive single-digit number.')
```

The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

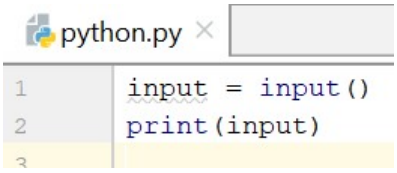


3.8 Keyboard input

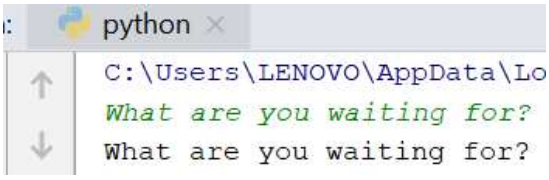
The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python provides a built-in function called input that gets input from the keyboard1. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

Code



Result



Chapter 4 Loops and Iteration

There are two loop methods in this capter namely For loop and While loop. The basic For loop use when one knows the number of iterations and use While loop when one doesn't know the number of iterations a priori [8].

4.1 For loop

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc. For example, print each fruit in a fruit list:

Code

```
python.py
1  fruits = ["apple", "banana", "cherry"]
2  for x in fruits:
3    print(x)
```

Result

```
python
C:\Users\
apple
banana
cherry
```

The for loop does not require an indexing variable to set beforehand.

4.1.1 Looping Through a String

Even strings are iterable objects, they contain a sequence of characters. For example, loop through the letters in the word "banana":

Code

```
for x in "banana":
  print(x)
```

Result

```
C:\Users
b
a
n
a
n
a
```

4.1.2 The break Statement

With the break statement we can stop the loop before it has looped through all the items.

For example, exit the loop when x is "banana":

Code

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

Result

```
C:\Users\
apple
banana
```

Example, exit the loop when x is "banana", but this time the break comes before the print:

Code

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

Result

```
C:\Users\
apple
```

4.1.3 The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next. Example, do not print banana:

Code

```
python.py
1  fruits = ["apple", "banana", "cherry"]
2  for x in fruits:
3    if x == "banana":
4      continue
5    print(x)
```

Result

```
python
C:\Users\LE
apple
cherry
```

4.1.4 The range() Function

To loop through a set of code a specified number of times, we can use the range() function. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. Example, using the range() function:

Code

```
python.py
1  for x in range(3):
2    print(x)
```

Result

```
python
C:\Users
0
1
2
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6).

Example, using the start parameter:

Code

```
for x in range(2, 6):
  print(x)
```

Result

```
C:\Users\I
2
3
4
5
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 11, 2).Example, increment the sequence with 3 (default is 1):

Code

```
for x in range(2, 11, 2):
  print(x)
```

Result

```
C:\Users\
2
4
6
8
10
```

4.1.5 Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished. Example print all numbers from 0 to 2, and print a message when the loop has ended:

Code

```
for x in range(3):
  print(x)
else:
  print("Finally finished!")
```

Result

```
C:\Users\LENOVO\A
0
1
2
Finally finished!
```

4.1.6 Nested Loops

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop". Example print each adjective for every fruit:

Code

```
python.py
1  adj = ["red", "big", "tasty"]
2  fruits = ["apple", "banana", "cherry"]
3
4  for x in adj:
5    for y in fruits:
6      print(x, y)
```

Result

```
python
C:\Users\LENOVO\
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

4.2 While loop

With the while loop we can execute a set of statements as long as a condition is true. For example, print i as long as i is less than 6:

Code

```
python.py
1  i = 1
2  while i < 6:
3    print(i)
4    i += 1
```

Result

```
python
C:\Users
1
2
3
4
5
```

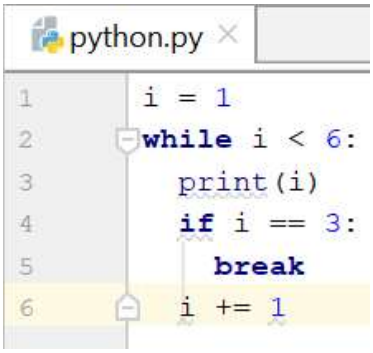
The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

4.2.1 The break Statement

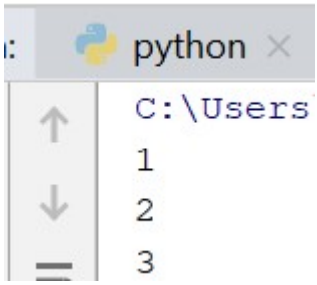
With the break statement we can stop the loop even if the while condition is true.

For example, exit the loop when i is 3:

Code



Result

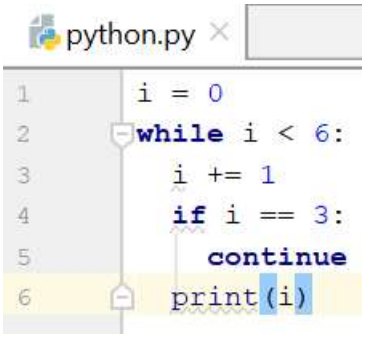


4.2.2 The continue Statement

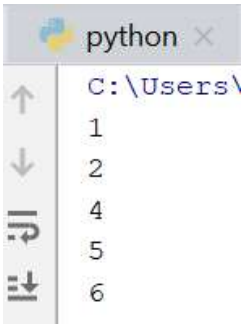
With the continue statement we can stop the current iteration and continue with the next.

For example, continue to the next iteration if i is 3:

Code



Result



4.2.3 The else Statement

With the else statement we can run a block of code once when the condition no longer is true. For example, print a message once the condition is false:

Code

```
python.py
1  i = 1
2  while i < 6:
3    print(i)
4    i += 1
5  else:
6    print("i is no longer less than 6")
```

Result

```
python
C:\Users\LENOVO\AppData\Loc
1
2
3
4
5
i is no longer less than 6
```

Chapter 5 Arrays

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

5.1 Lists

A list is a collection which is ordered and changeable. In Python lists are written with square brackets. Example, create a list:

Code

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

Resutl

```
C:\Users\LENOVO\AppData\Local\
['apple', 'banana', 'cherry']
```

5.1.1 Access Items

You access the list items by referring to the index number. Example, print the second item of the list:

Code

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Result

```
C:\Users\
banana
```

5.1.2 Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc. Example, print the last item of the list:

Code

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Result

```
C:\Users
cherry
```

5.1.3 Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items. Example, return the third, fourth, and fifth item:

Code

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"
print(thislist[2:5])
```

Result

```
C:\Users\LENOVO\AppData\Local
['cherry', 'orange', 'kiwi']
```

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

5.1.4 Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list. Example, this example returns the items from index -4 (included) to index -1 (excluded)

Code

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

Result

```
C:\Users\LENOVO\AppData\Loc
['orange', 'kiwi', 'melon']
```

5.1.5 Change Item Value

To change the value of a specific item, refer to the index number. Example, change the second item:

Code

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Result

```
C:\Users\LENOVO\AppData\Local\Progr
['apple', 'blackcurrant', 'cherry']
```

5.1.6 Loop Through a List

You can loop through the list items by using a for loop. Example, print all items in the list, one by one:

Code

Result

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

```
C:\User:
apple
banana
cherry
```

5.1.7 Check if Item Exists

To determine if a specified item is present in a list use the in keyword. Example, check if "apple" is present in the list:

Code

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Result

```
C:\Users\LENOVO\AppData\Local\Prog:
Yes, 'apple' is in the fruits list
```

5.1.8 List Length

To determine how many items a list has, use the len() method. Example, print the number of items in the list:

Code

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Result

```
C:\U:
3
```

5.1.9 Add Items

To add an item to the end of the list, use the append() method. Example, using the append() method to append an item:

Code

Result

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

```
C:\Users\LENOVO\AppData\Local\Programs\
['apple', 'banana', 'cherry', 'orange']
```

To add an item at the specified index, use the insert() method. Example, insert an item as the second position:

Code

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Result

```
C:\Users\LENOVO\AppData\Local\Programs\I
['apple', 'orange', 'banana', 'cherry']
```

5.1.10 Remove Item

There are several methods to remove items from a list. Example, the remove() method removes the specified item:

Code

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Result

```
C:\Users\LENOVO\Appl
['apple', 'cherry']
```

Example, the pop() method removes the specified index, (or the last item if index is not specified):

Code

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Result

```
C:\Users\LENOVO\Appl
['apple', 'banana']
```

Example, the del keyword removes the specified index:

Code

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

Result

```
C:\Users\LENOVO\AppD
['banana', 'cherry']
```

Example, the del keyword can also delete the list completely:

Code

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist)
```

Result

```
    print(thislist)
NameError: name 'thislist' is not defined
```

Example, the clear() method empties the list:

Code

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Result

```
C:\Users
[]
```

5.1.11 Copy a List

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2. There are ways to make a copy, one way is to use the built-in List method copy(). Example, make a copy of a list with the copy() method:

Code

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Result

```
C:\Users\LENOVO\AppData\Local\
['apple', 'banana', 'cherry']
```

Another way to make a copy is to use the built-in method list(). Example, make a copy of a list with the list() method:

Code

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Result

```
C:\Users\LENOVO\AppData\Local'
['apple', 'banana', 'cherry']
```

5.1.12 Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python. One of the easiest ways are by using the + operator. Example, join two list:

Code

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

Result

```
C:\Users\LENOVO\AppData\
['a', 'b', 'c', 1, 2, 3]
```

Another way to join two lists are by appending all the items from list2 into list1, one by one. Example, append list2 into list1:

Code

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

Result

```
C:\Users\LENOVO\AppData\
['a', 'b', 'c', 1, 2, 3]
```

Or you can use the extend() method, which purpose is to add elements from one list to another list. Example, use the extend() method to add list2 at the end of list1:

Code

Result

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

```
C:\Users\LENOVO\AppData\
['a', 'b', 'c', 1, 2, 3]
```

5.1.13 The list() Constructor

It is also possible to use the list() constructor to make a new list. Example, using the list() constructor to make a List:

Code

```
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```

Result

```
C:\Users\LENOVO\AppData\Local'
['apple', 'banana', 'cherry']
```

5.2 Tuple

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets. Example, create a Tuple:

Code

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Result

```
C:\Users\LENOVO\AppData\Local
('apple', 'banana', 'cherry')
```

5.2.1 Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets. Example, print the second item in the tuple:

Code

Result

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

```
C:\Users\
banana
```

5.2.2 Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc. Example, print the last item of the tuple:

Code

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Result

```
C:\Users\
cherry
```

5.2.3 Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items. Example, return the third, fourth, and fifth item:

Code

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Result

```
C:\Users\LENOVO\AppData\Loca
('cherry', 'orange', 'kiwi')
```

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

5.2.4 Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple. Example, this example returns the items from index -4 (included) to index -1 (excluded)

Code

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Result

```
C:\Users\LENOVO\AppData\Loc
('orange', 'kiwi', 'melon')
```

5.2.5 Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called. But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple. Example, convert the tuple into a list to be able to change it:

Code

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Result

```
C:\Users\LENOVO\AppData\Loc
('apple', 'kiwi', 'cherry')
```

5.2.6 Loop Through a Tuple

You can loop through the tuple items by using a for loop. Example, iterate through the items and print the values:

Code

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Result

```
C:\Users\
apple
banana
cherry
```

5.2.7 Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword. Example, check if "apple" is present in the tuple:

Code

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Result

```
C:\Users\LENOVO\AppData\Local\Progra
Yes, 'apple' is in the fruits tuple
```

5.2.8 Tuple Length

To determine how many items a tuple has, use the len() method. Example, print the number of items in the tuple:

Code Result

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

```
C:\U
3
```

5.2.9 Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**. Example, you cannot add items to a tuple:

Code

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Result

```
  thistuple[3] = "orange" # This will raise an error
TypeError: 'tuple' object does not support item assignment
```

5.2.10 Create Tuple With One Item

To create a tuple with only one item, you have add a comma after the item, unless Python will not recognize the variable as a tuple. Example, one item tuple, remember the commma:

Code

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Result

```
C:\Users\LENOVO\
<class 'tuple'>
<class 'str'>
```

5.2.11 Remove Items

Note: You cannot remove items in a tuple. Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely. Example, the del keyword can delete the tuple completely:

Code

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Result

```
    print(thistuple) #this will raise an error because the tuple no longer exists
NameError: name 'thistuple' is not defined
```

5.2.12 Join Two Tuples

To join two or more tuples you can use the + operator. Example, join two tuples:

Code

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Result

```
C:\Users\LENOVO\AppData\
('a', 'b', 'c', 1, 2, 3)
```

5.2.13 The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple. Example, using the tuple() method to make a tuple:

Code

```
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

Result

```
C:\Users\LENOVO\AppData\Local
('apple', 'banana', 'cherry')
```

5.3 Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets. Example, create a Set:

Code

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Result

```
C:\Users\LENOVO\AppData\Local\
{'apple', 'banana', 'cherry'}
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

5.3.1 Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword. Example, loop through the set, and print the values:

Code

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Result

```
C:\Users
banana
apple
cherry
```

Example, check if "banana" is present in the set:

Code

Result

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

```
C:\Users\
True
```

5.3.2 Change Items

Once a set is created, you cannot change its items, but you can add new items.

5.3.3 Add Items

To add one item to a set use the add() method. To add more than one item to a set use the update() method. Example, add an item to a set, using the add() method:

Code

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Result

```
C:\Users\LENOVO\AppData\Local\Programs\
{'apple', 'orange', 'banana', 'cherry'}
```

Example, add multiple items to a set, using the update() method:

Code

```
thisset = {"apple", "banana", "cherry"}
thisset.update(["orange", "mango", "grapes"])
print(thisset)
```

Result

```
{'orange', 'banana', 'apple', 'grapes', 'mango', 'cherry'}
```

5.3.4 Get the Length of a Set

To determine how many items a set has, use the len() method. Example, get the number of items in a set:

Code | Result

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

Result: 3

5.3.5 Remove Item

To remove an item in a set, use the remove(), or the discard() method. Example, remove "banana" by using the remove() method:

Code | Result

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Result: {'apple', 'cherry'}

Note: If the item to remove does not exist, remove() will raise an error.

Example, remove "banana" by using the discard() method:

Code | Result

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Result: {'cherry', 'apple'}

Note: If the item to remove does not exist, discard() will NOT raise an error.

You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed. The return value of the pop() method is the removed item. Example, remove the last item by using the pop() method:

Code | Result

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

```
banana
{'cherry', 'apple'}
```

Note: Sets are unordered, so when using the pop() method, you will not know which item that gets removed. Example, the clear() method empties the set:

Code

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

Result

```
set()
```

Example, the del keyword will delete the set completely:

Code

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

Result

```
    print(thisset)
NameError: name 'thisset' is not defined
```

5.3.6 Join Two Sets

There are several ways to join two or more sets in Python. You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another. Example, the union() method returns a new set with all items from both sets:

Code

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

Result

```
{'b', 'a', 1, 2, 'c', 3}
```

Example, the update() method inserts the items in set2 into set1:

Code

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

Result

```
{1, 2, 3, 'a', 'b', 'c'}
```

Note: Both union() and update() will exclude any duplicate items. There are other methods that joins two sets and keeps ONLY the duplicates, or NEVER the duplicates, check the full list of set methods in the bottom of this page.

5.3.7 The set() Constructor

It is also possible to use the set() constructor to make a set. Example, using the set() constructor to make a set:

Code

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

Result

```
{'banana', 'cherry', 'apple'}
```

5.4 Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. Example, create and print a dictionary:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Result

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

5.4.1 Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets. Example, get the value of the "model" key:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)

x = thisdict["model"]
print(x)
```

Result

```
{'model': 'Mustang', 'brand': 'Ford', 'year': 1964}
Mustang
```

There is also a method called get() that will give you the same result. Example, get the value of the "model" key:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)

x = thisdict["model"]
print(x)

x = thisdict.get("model")
print(x)
```

Result

```
{'year': 1964, 'model': 'Mustang', 'brand': 'Ford'}
Mustang
Mustang
```

5.4.2 Change Values

You can change the value of a specific item by referring to its key name. Example, change the "year" to 2018:

Code Result

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
print(thisdict["year"])
```

```
2018
```

5.4.3 Loop Through a Dictionary

You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well. Example, print all key names in the dictionary, one by one:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
```

Result

```
brand
year
model
```

Example, print all values in the dictionary, one by one:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(thisdict[x])
```

Result

```
Ford
1964
Mustang
```

Example, you can also use the values() function to return values of a dictionary:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
  print(x)
```

Result

```
Ford
1964
Mustang
```

Example, loop through both keys and values, by using the items() function:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

Result

```
year 1964
brand Ford
model Mustang
```

5.4.4 Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword. Example, check if "model" is present in the dictionary:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Result

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

5.4.5 Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the len() method.

Example, print the number of items in the dictionary:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(len(thisdict))
```

Result

```
3
```

5.4.6 Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Result

```
{'color': 'red', 'model': 'Mustang', 'brand': 'Ford', 'year': 1964}
```

5.4.7 Removing Items

There are several methods to remove items from a dictionary. Example, the pop() method removes the item with the specified key name:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Result

```
{'brand': 'Ford', 'year': 1964}
```

Example, the popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Result

```
{'model': 'Mustang', 'year': 1964}
```

Example, the del keyword removes the item with the specified key name:

Code

Result

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'year': 1964, 'brand': 'Ford'}
```

Example, the del keyword can also delete the dictionary completely:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Result

```
    print(thisdict) #this will cause an error because "thisdict" no longer exists.
NameError: name 'thisdict' is not defined
```

Example, the clear() keyword empties the dictionary:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Result

```
{}
```

5.4.8 Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2. There are ways to make a copy, one way is to use the built-in Dictionary method copy(). Example, make a copy of a dictionary with the copy() method:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Result

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Another way to make a copy is to use the built-in method dict(). Example, make a copy of a dictionary with the dict() method:

Code

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Result

```
{'model': 'Mustang', 'brand': 'Ford', 'year': 1964}
```

5.4.9 Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries. Example, create a dictionary that contain three dictionaries:

Code

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
print(myfamily)
```

Result

```
{'child1': {'year': 2004, 'name': 'Emil'}, 'child3': {'year': 2011, 'name': 'Linus'}, 'child2': {'year': 2007, 'name': 'Tobias'}}
```

Or, if you want to nest three dictionaries that already exists as dictionaries. Example, create three dictionaries, than create one dictionary that will contain the other three dictionaries:

Code

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
print(myfamily)
```

Result

```
{'child1': {'year': 2004, 'name': 'Emil'}, 'child2': {'year': 2007, 'name': 'Tobias'}, 'child3': {'year': 2011, 'name': 'Linus'}}
```

5.4.10 The dict() Constructor

It is also possible to use the dict() constructor to make a new dictionary:

Code

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Result

```
{'model': 'Mustang', 'brand': 'Ford', 'year': 1964}
```

Chapter 6 Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

6.1 Creating a Function

In Python a function is defined using the def keyword:

```
def my_function():
  print("Hello from a function")
```

6.2 Calling a Function

To call a function, use the function name followed by parenthesis:

Code

```
def my_function():
  print("Hello from a function")

my_function()
```

Result

```
Hello from a function
```

6.3 Parameters

Information can be passed to functions as parameter. Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma. The following example has a function with one parameter (fname).

When the function is called, we pass along a first name, which is used inside the function to print the full name:

Code

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Result

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

6.4 Default Parameter Value

The following example shows how to use a default parameter value. If we call the function without parameter, it uses the default value:

Code

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Result

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

6.5 Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function. E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

Code

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

my_function(fruits)
```

Result

```
apple
banana
cherry
```

6.6 Return Values

To let a function return a value, use the return statement:

Code

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Result

```
15
25
45
```

6.7 Keyword Arguments

You can also send arguments with the key = value syntax. This way the order of the arguments does not matter.

Code

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Result

```
The youngest child is Linus
```

The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

6.8 Arbitrary Arguments

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a tuple of arguments and can access the items accordingly. Example, if the number of arguments is unknown, add a * before the parameter name:

Code

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Result

```
The youngest child is Linus
```

6.9 Recursion

Python also accepts function recursion, which means a defined function can call itself. Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result. The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming. In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0). To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Code

```
def tri_recursion(k):
  if(k>0):
    result = k+tri_recursion(k-1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Result

```
Recursion Example Results
1
3
6
10
15
21
```

Chapter 7 Files Handling

File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files.

7.1 File Handling

The key function for working with files in Python is the open() function. The open() function takes two parameters; filename, and mode. There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

7.1.1 Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

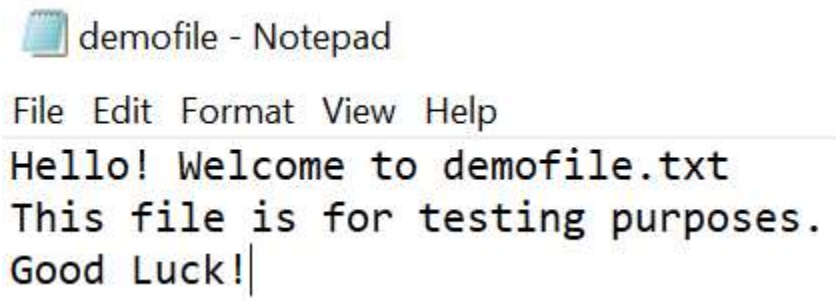
Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

7.2 Read File

7.2.1 Open a File on the Server

Assume we have the following file, located in the same folder as Python:



To open the file, use the built-in open() function. The open() function returns a file object,

which has a read() method for reading the content of the file:

Code

```
f = open("demofile.txt", "r")
print(f.read())
```

Result

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

7.2.2 Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return. Example, return the 5 first characters of the file:

Code

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Result

```
Hello
```

7.2.3 Read Lines

You can return one line by using the readline() method. Example, read one line of the file:

Code

```
f = open("demofile.txt", "r")
print(f.readline())
```

Result

```
Hello! Welcome to demofile.txt
```

By calling readline() two times, you can read the two first lines. Example, read two lines of the file:

Code

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Result

```
Hello! Welcome to demofile.txt

This file is for testing purposes.
```

By looping through the lines of the file, you can read the whole file, line by line. Example, loop through the file line by line:

Code

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Result

```
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!
```

7.2.4 Close Files

It is a good practice to always close the file when you are done with it. Example, close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

7.3 Write/Create File

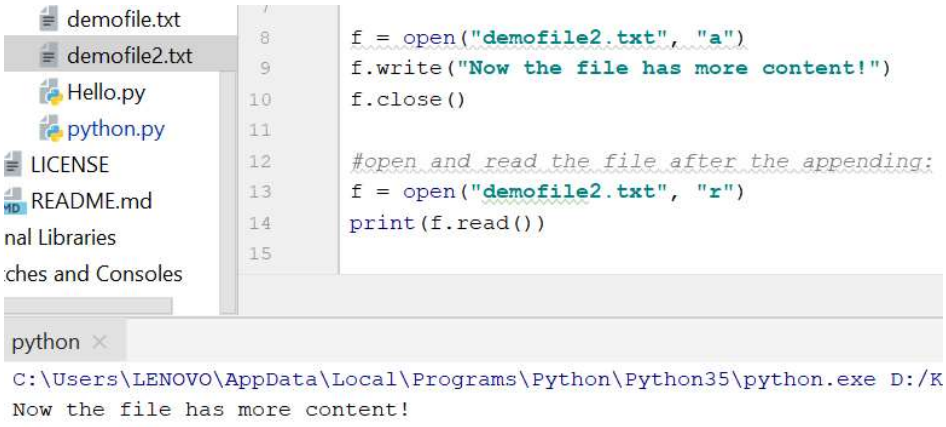
7.3.1 Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

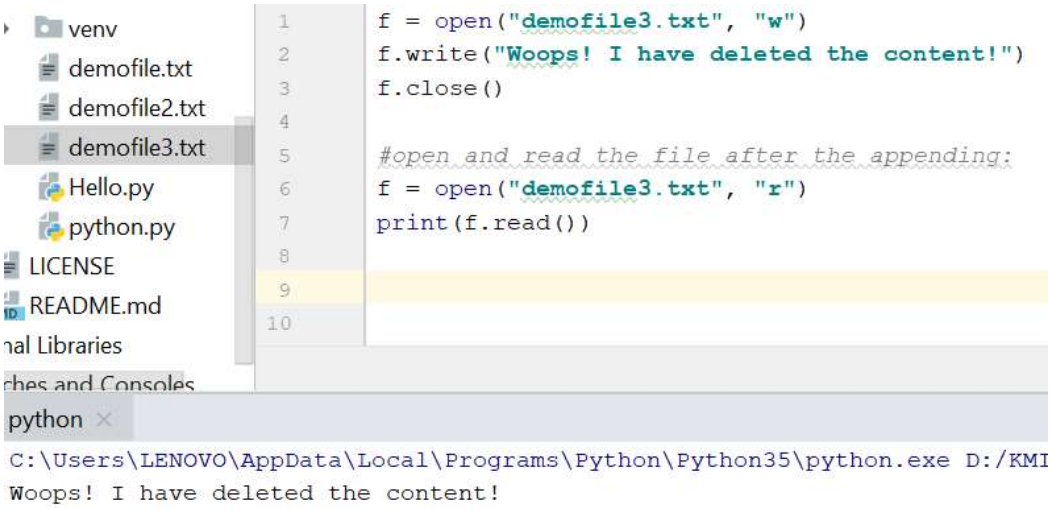
"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example, open the file "demofile2.txt" and append content to the file:



Example, open the file "demofile3.txt" and overwrite the content:



Note: the "w" method will overwrite the entire file.

7.3.2 Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example, create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created! Example, create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

7.4 Delete a File

To delete a file, you must import the OS module, and run its os.remove() function. Example,

remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

7.4.1 Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it. Example, check if file exists, then delete it:

Code

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Result

```
The file does not exist
```

7.4.2 Delete Folder

To delete an entire folder, use the os.rmdir() method. Example, remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove empty folders.

Python Cheat Sheet

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

Python For Data Science *Cheat Sheet*

Python Basics

Learn More Python for Data Science Interactively at www.datacamp.com

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
5
```

Calculations With Variables

```
>>> x+2
7
>>> x-2
3
>>> x*2
10
>>> x**2
25
>>> x%2
1
>>> x/float(2)
2.5
```

Sum of two variables
Subtraction of two variables
Multiplication of two variables
Exponentiation of a variable
Remainder of a variable
Division of a variable

Types and Type Conversion

`str()`	`'5', '3.45', 'True'`	Variables to strings
`int()`	`5, 3, 1`	Variables to integers
`float()`	`5.0, 1.0`	Variables to floats
`bool()`	`True, True, True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

```
>>> my_list[1]
>>> my_list[-3]
```

Select item at index 1
Select 3rd last item

Slice

```
>>> my_list[1:3]
>>> my_list[1:]
>>> my_list[:3]
>>> my_list[:]
```

Select items at index 1 and 2
Select items after index 0
Select items before index 3
Copy my_list

Subset Lists of Lists

```
>>> my_list2[1][0]
>>> my_list2[1][:2]
```

my_list[list][itemOfList]

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis
learn — Machine learning
NumPy — Scientific computing
matplotlib — 2D plotting

Install Python

Leading open data science platform powered by Python

Free IDE that is included with Anaconda

Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset

```
>>> my_array[1]
  2
```

Select item at index 1

Slice

```
>>> my_array[0:2]
  array([1, 2])
```

Select items at index 0 and 1

Subset 2D Numpy arrays

```
>>> my_2darray[:,0]
  array([1, 4])
```

my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3
  array([False, False, False,  True], dtype=bool)
>>> my_array * 2
  array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
  array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

DataCamp
Learn Python for Data Science Interactively

References

[1] [Online]. Available: http://www.pythonlearn.com/html-007/cfbook002.html.

[2] [Online]. Available: https://www.softwaretestinghelp.com/python-ide-code-editors/.

[3] [Online]. Available: https://www.jetbrains.com/pycharm/download/#section=windows.

[4] [Online]. Available: https://www.python.org/downloads/.

[5] [Online]. Available: https://www.w3schools.com/python/.

[6] [Online]. Available: https://desktop.github.com/.

[7] [Online]. Available: https://programminghistorian.org/en/lessons/getting-started-with-github-desktop#what-are-git-and-github.

[8] [Online]. Available: https://www.w3schools.com/python/python_for_loops.asp.

Exercises & Solution