

Teaching materials

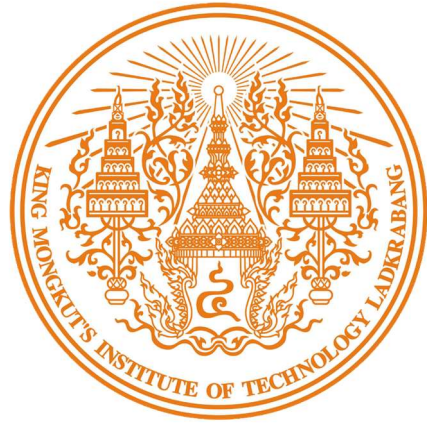
15016406: COMPUTER PROGRAMMING

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

Chapter 1 Why we Program	1
1.1 Installing and Using Python	2
1.1.1 Python Editor	2
1.1.2 Python executes(compiler)	8
1.2 The first program	9
1.3 Installing and Using GitHub.....	12
1.3.1 GitHub Introduction	14
1.3.2 Version Control Using GitHub Desktop	15
1.3.3 Using GitHub and GitHub Desktop.....	18
Chapter 2 Variables, expressions and statements.....	27
2.1 Values and types	27
2.2 Variables	29
2.3 Variable names and keywords.....	31
2.4 Statements.....	32
2.5 Operators and operands	33
2.6 Expressions.....	34
2.7 Order of operations.....	35

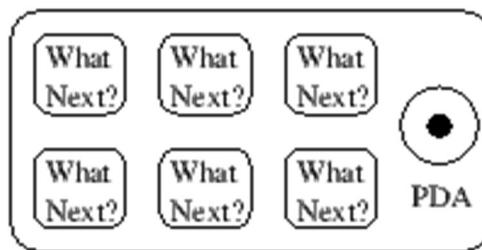
2.8 String operations	36
2.9 Comments.....	37
Chapter 3 Conditionals	38
3.1 Modulus operator.....	38
3.2 Boolean expressions	38
3.3 Logical operators	39
3.4 Conditional execution	40
3.5 Alternative execution.....	41
3.6 Chained conditionals	41
3.7 Nested conditionals	42
3.8 Keyboard input	44
Chapter 4 Loops and Iteration	45
4.1 For loop.....	45
4.1.1 Looping Through a String.....	45
4.1.2 The break Statement.....	46
4.1.3 The continue Statement.....	47
4.1.4 The range() Function.....	47
4.1.5 Else in For Loop.....	48

4.1.6 Nested Loops	49
4.2 While loop	49
4.2.1 The break Statement.....	50
4.2.2 The continue Statement.....	50
4.2.3 The else Statement	51
Chapter 5 List and Dictionaries.....	52
Chapter 6 Functions	53
Chapter 7 Files.....	54
Python Cheat Sheet.....	55
References	57
Exercises & Solution.....	58

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 Editor

1. Python IDEs and Code Editors

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.



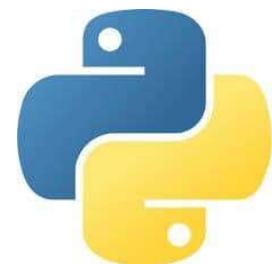
PyCharm



Spyder



Pydev



Idle



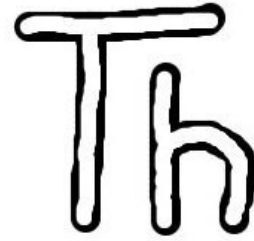
Wing



Eric Python



Rodeo



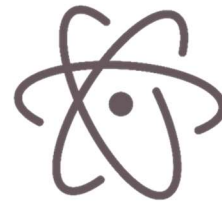
Thonny

2. Python IDEs and Code Editors

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.



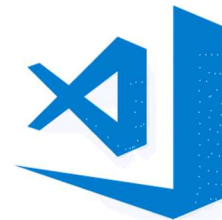
Sublime Text



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:

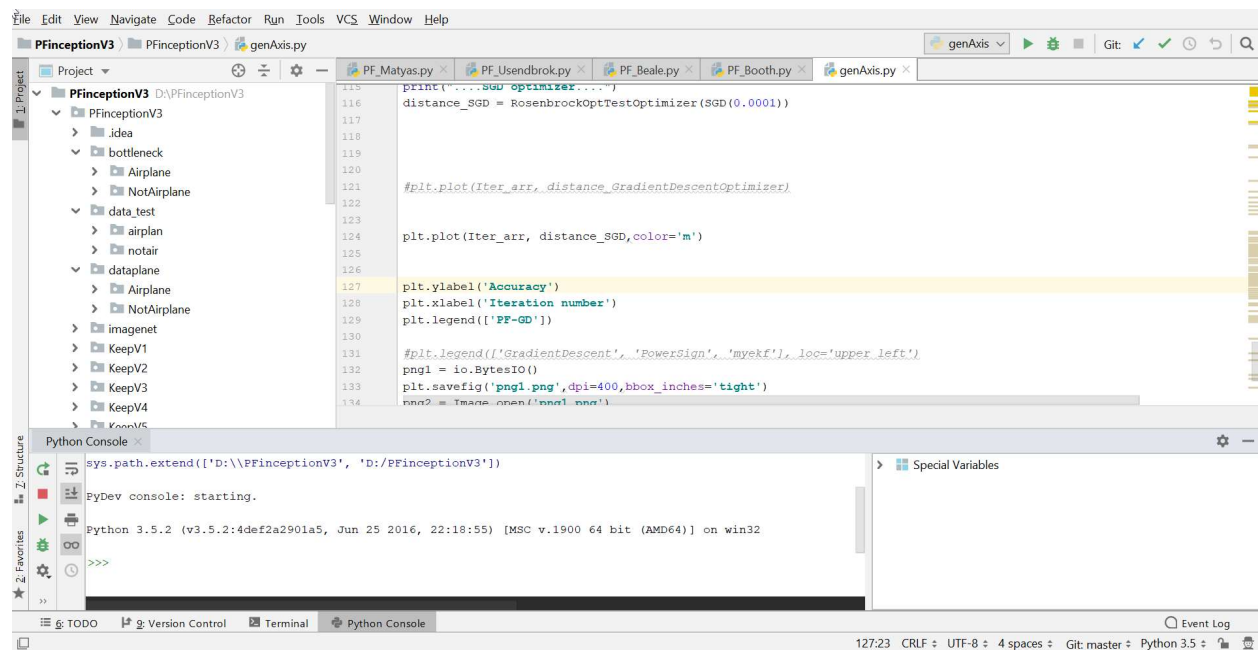
1.It comes with an intelligent code editor, smart code navigation, fast and safe refactoring's.

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

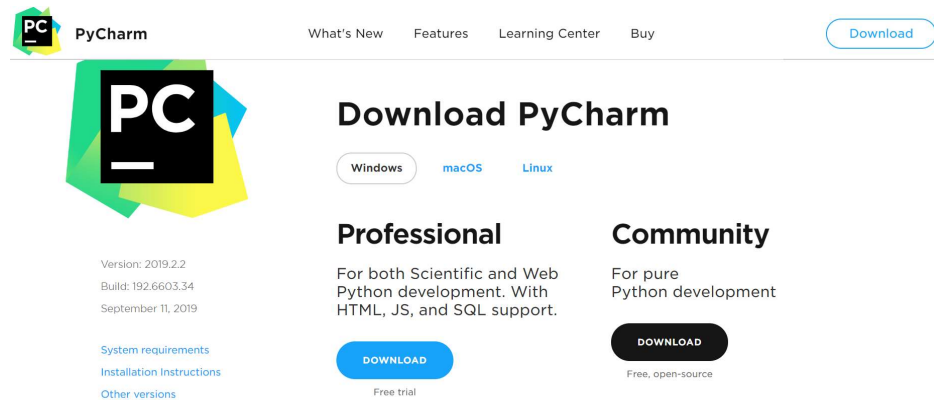
3.With Python, PyCharm also provides support to python web development frameworks, JavaScript, HTML, CSS, Angular JS and Live edit features.

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

Pros:



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

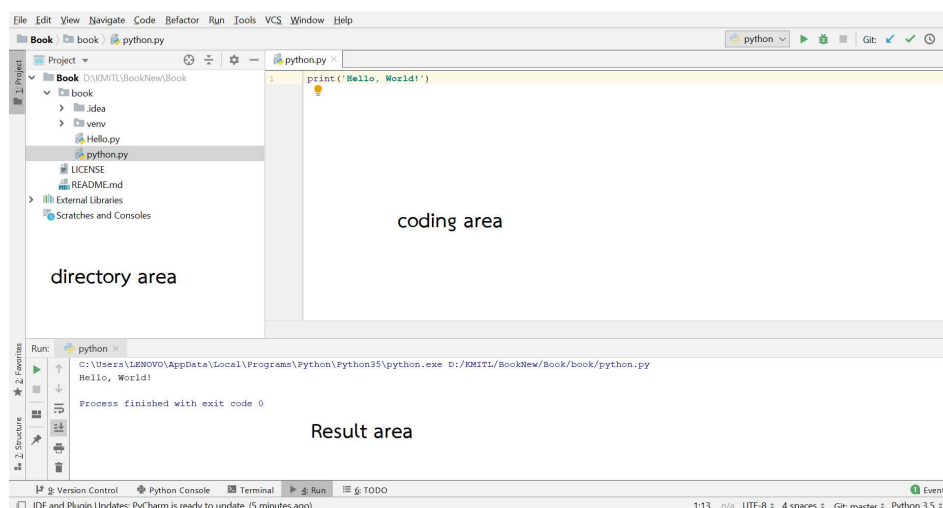


For beginner, the Community version is enough for learning Python Programming. The next section is about Python executes or compiler.

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?

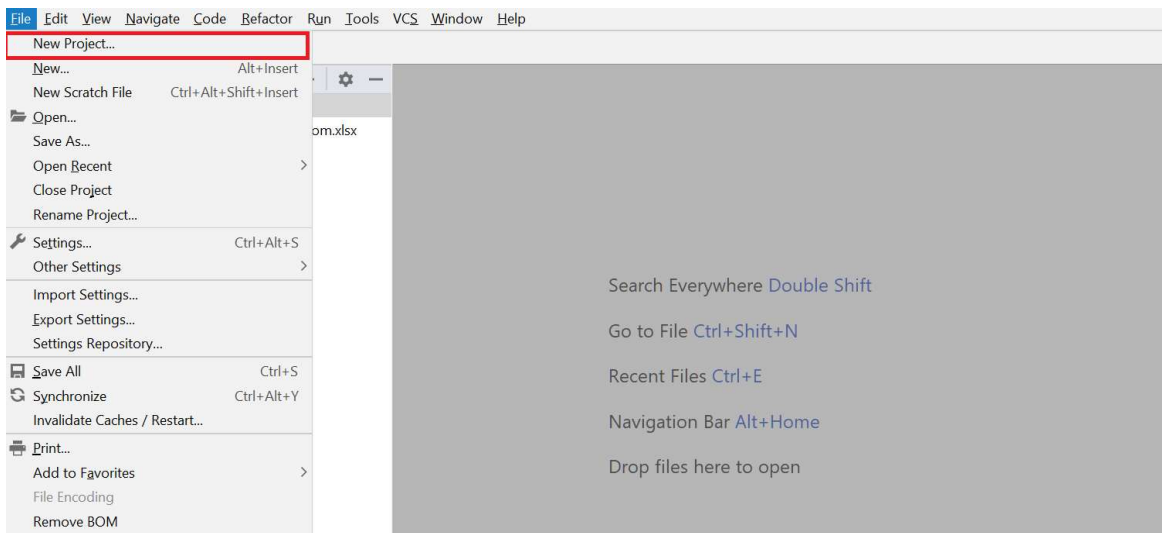
Python releases by version number:

Release version	Release date	Download	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

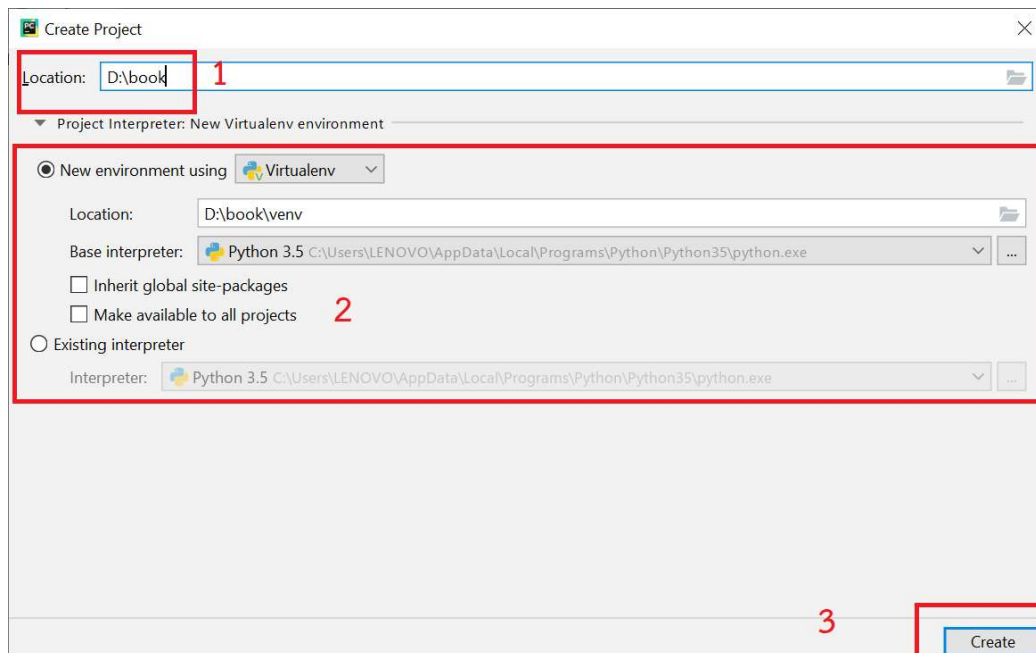
[View older releases](#)

1.2 The first program

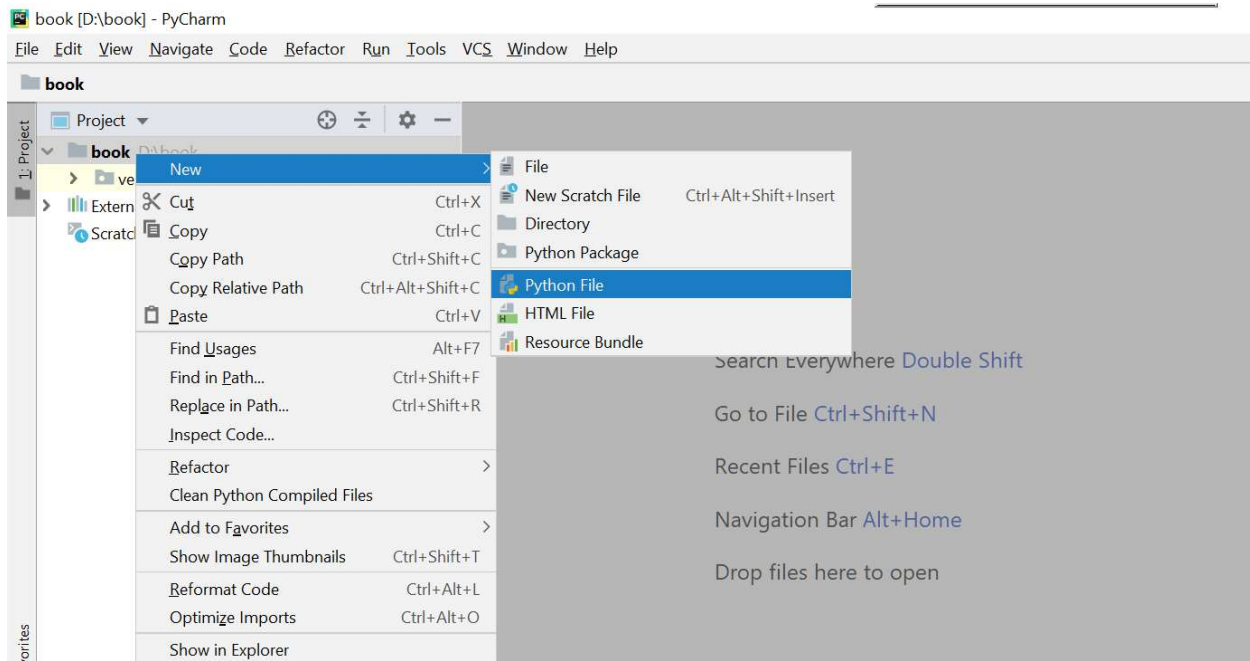
After install Python IDE and Python compiler, try to print sentence “Hello World” as the following code and suggestion. New Project as the below picture:



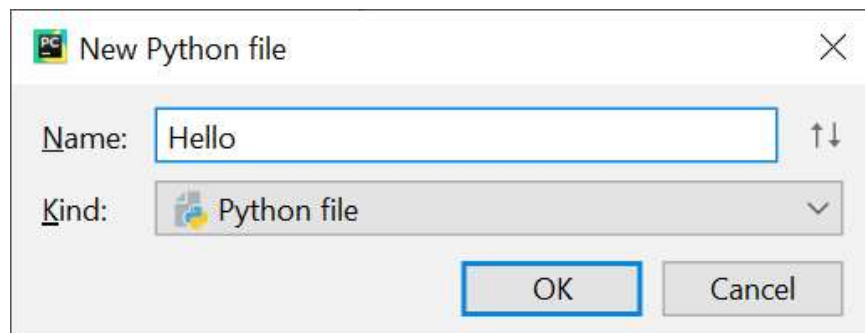
Set location for the new project, then select an interpreter by choosing the new environment or existing interpreter and create in the last step.



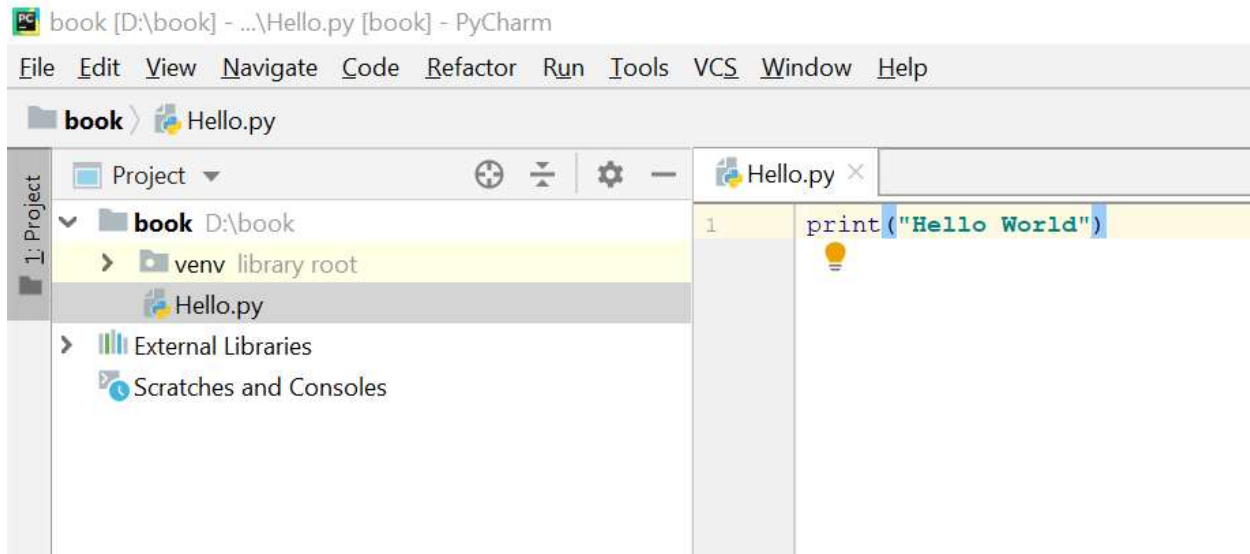
New Python File as the follow:



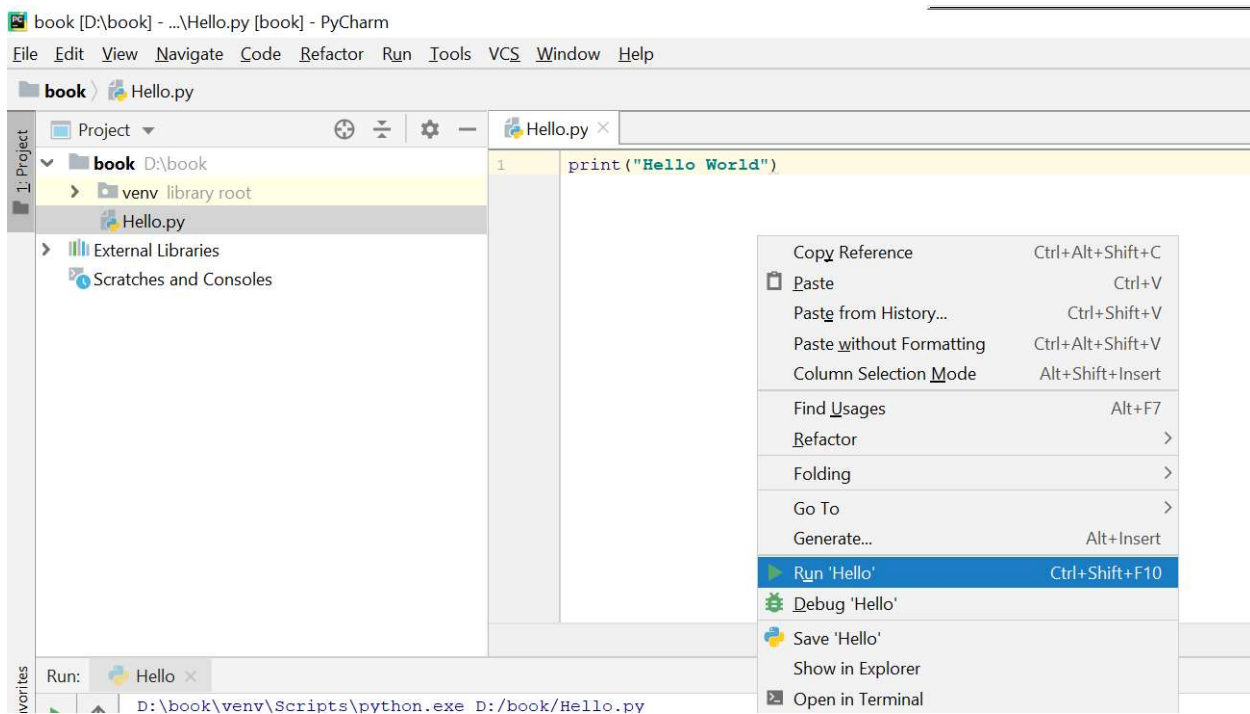
Type file name and then hit “OK”

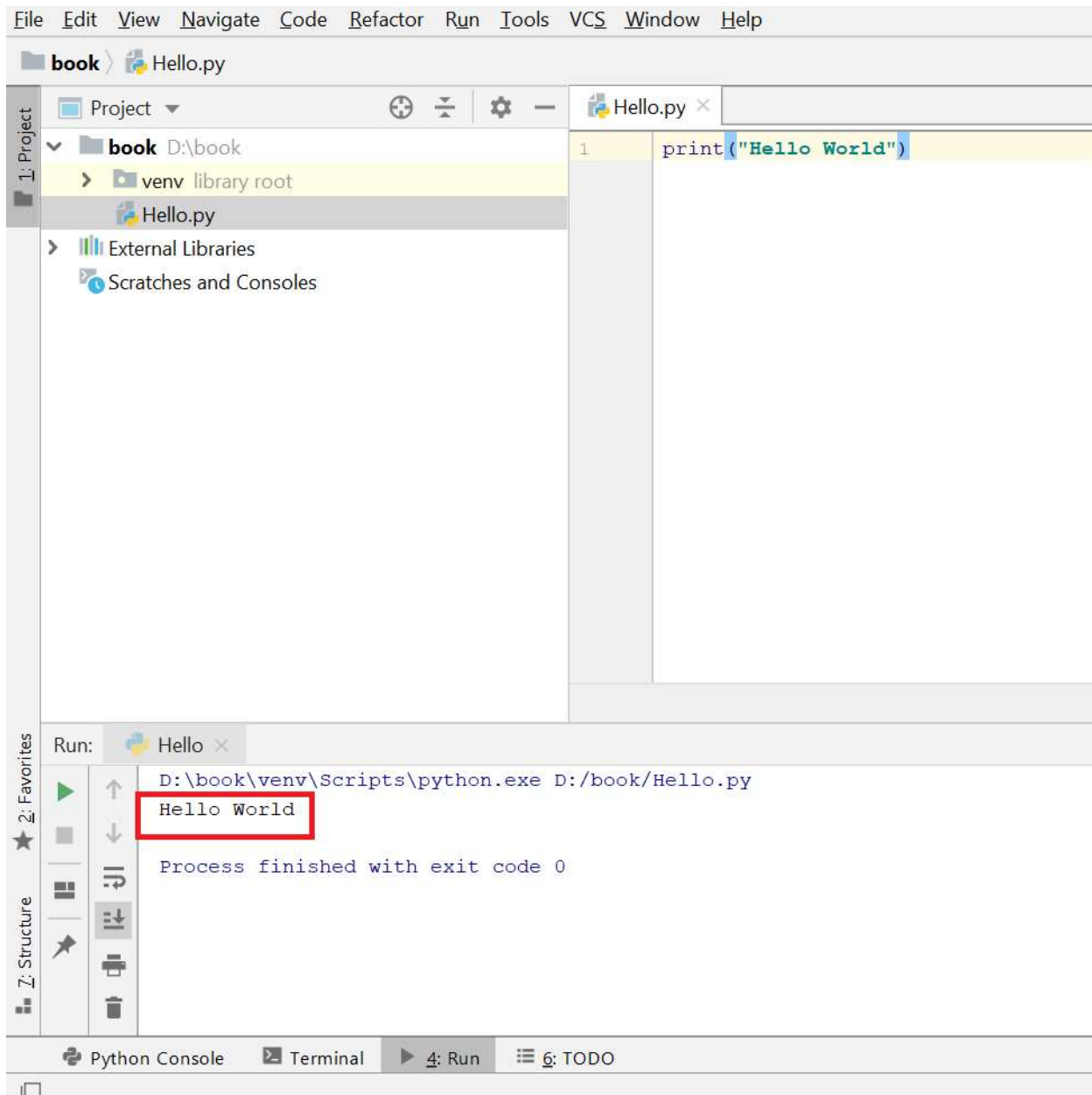


Type the first syntax => print(...) as following image



Right click on screen and select Run “youfilename”



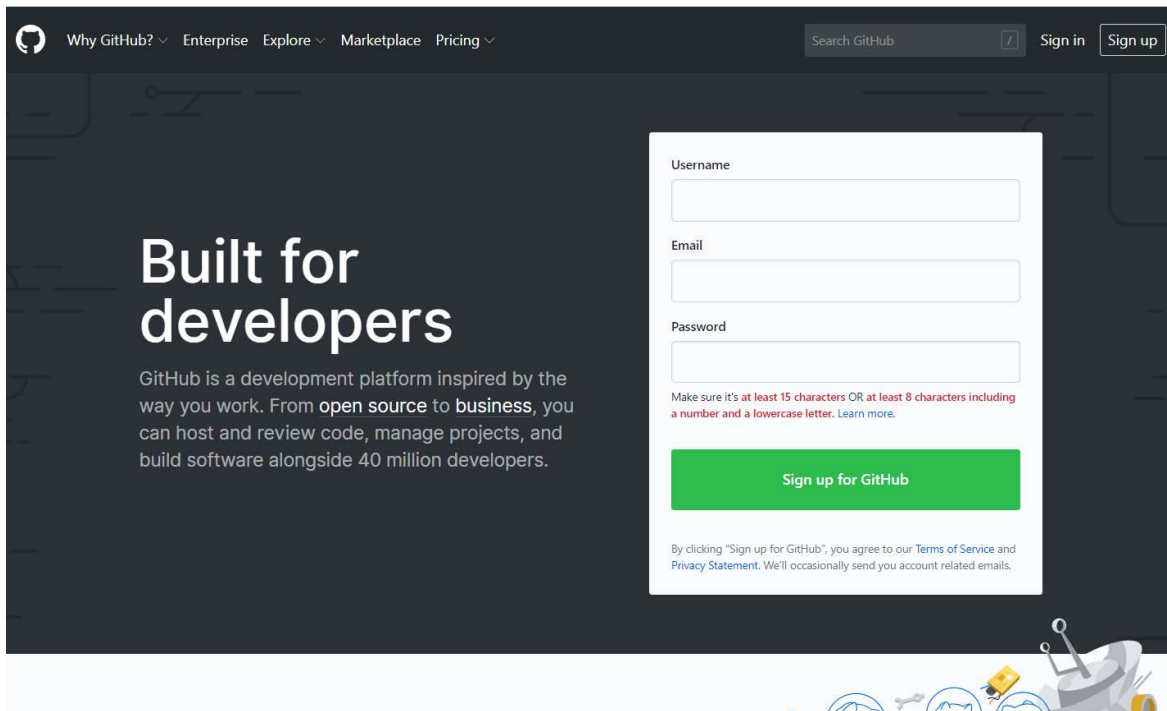


The result will show as follow with the word “Hello World”

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 [5] are implemented.



The image shows the GitHub homepage with a dark background. On the left, the text "Built for developers" is prominently displayed. Below it, a paragraph describes GitHub as a development platform. On the right, there is a white sign-up form with fields for Username, Email, and Password. A green button labeled "Sign up for GitHub" is at the bottom of the form. The top navigation bar includes links for "Why GitHub?", "Enterprise", "Explore", "Marketplace", and "Pricing", along with a search bar and "Sign in" and "Sign up" buttons.

Why GitHub? ▾ Enterprise ▾ Explore ▾ Marketplace ▾ Pricing ▾

Search GitHub [7] Sign in Sign up

Built for developers

GitHub is a development platform inspired by the way you work. From **open source** to **business**, you can host and review code, manage projects, and build software alongside 40 million developers.

Username

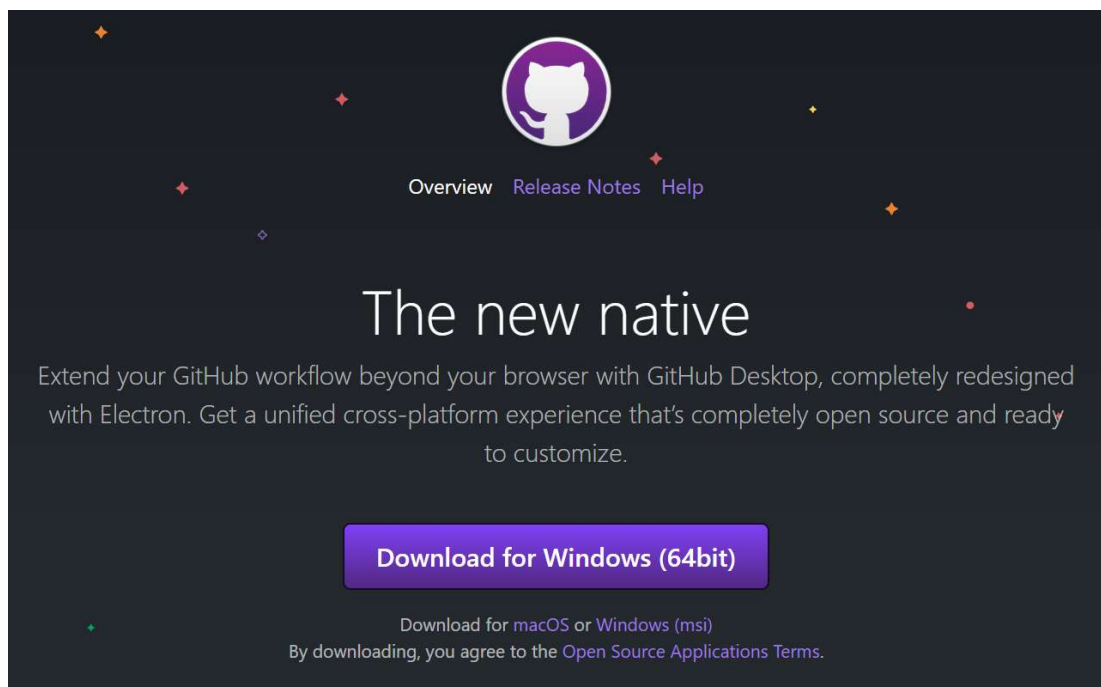
Email

Password

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)

Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our [Terms of Service](#) and [Privacy Statement](#). We'll occasionally send you account related emails.



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 [6].

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.

The screenshot displays the GitHub interface for the repository `tomtomAnalytics / iaai.asia`. The repository has 849 commits, 2 branches, 0 releases, 1 environment, and 1 contributor. The file list shows the following files and their commit history:

File	Commit Message	Commit Date
<code>contactform</code>	first	3 months ago
<code>css</code>	first	3 months ago
<code>img</code>	img	6 days ago
<code>js</code>	first	3 months ago
<code>letter</code>	update	21 days ago
<code>lib</code>	Revert "Update bootstrap.min.css"	2 months ago
<code>Accommodation.html</code>	update	6 days ago

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.

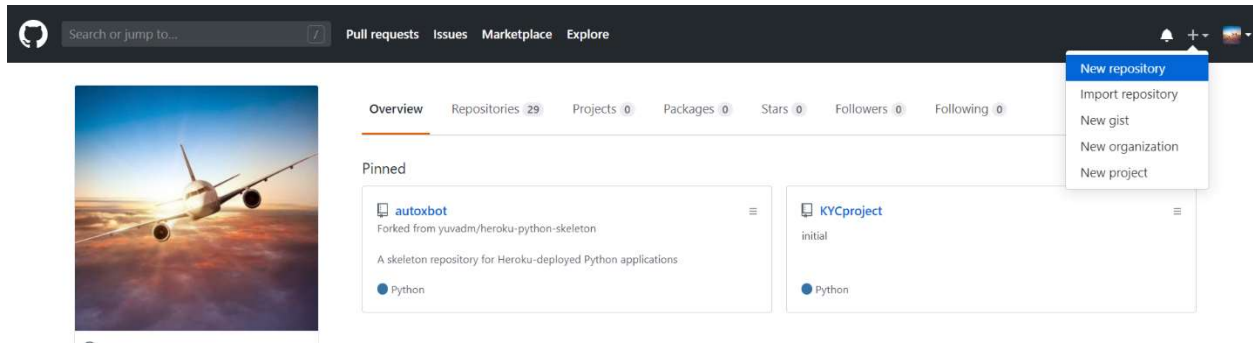
Register for a GitHub Account

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

tomtomAnalytics / Book

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Security Insights Settings

No description, website, or topics provided. [Edit](#)

[Manage topics](#)

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find File Clone or download

tomtomAnalytics Initial commit Latest commit 13468d3 now

LICENSE	Initial commit	now
README.md	Initial commit	now

README.md

Book

Clone Repository to Local by GitHub Desktop

File Edit View Repository Branch Help

New repository... Ctrl+N

Add local repository... Ctrl+O

Clone repository... Ctrl+Shift+O

Options... Ctrl+,

Exit Alt+F4

Current branch gh-pages Fetch origin Last fetched a day ago

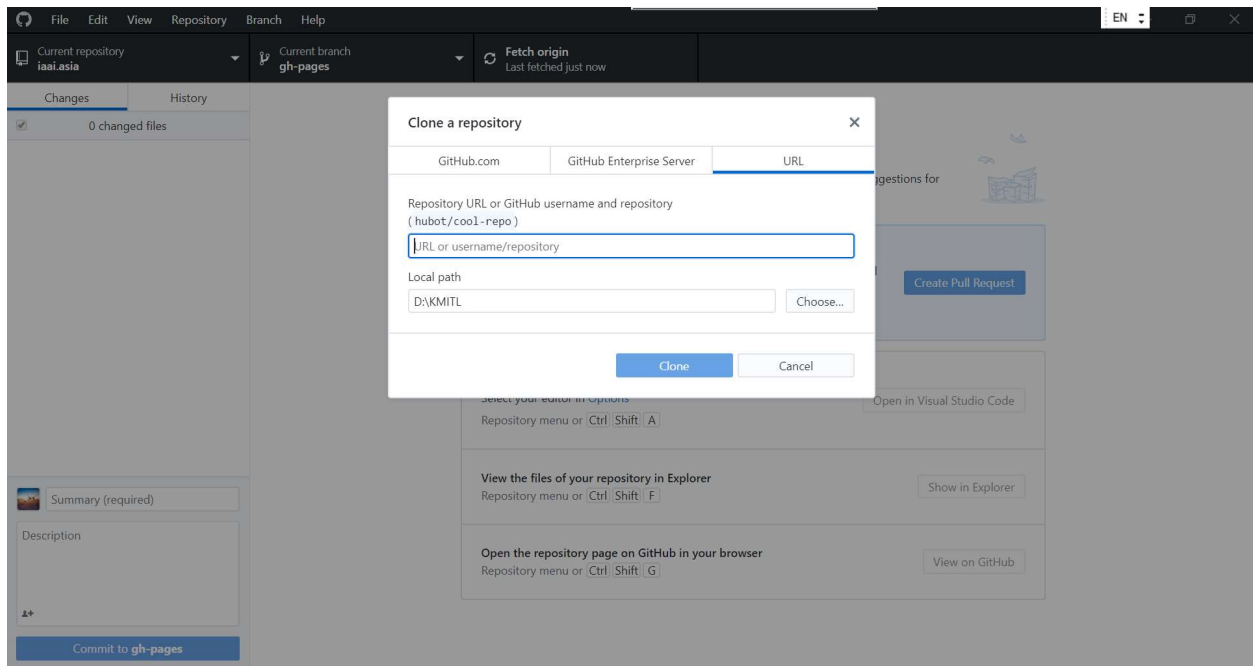
No local changes

There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.

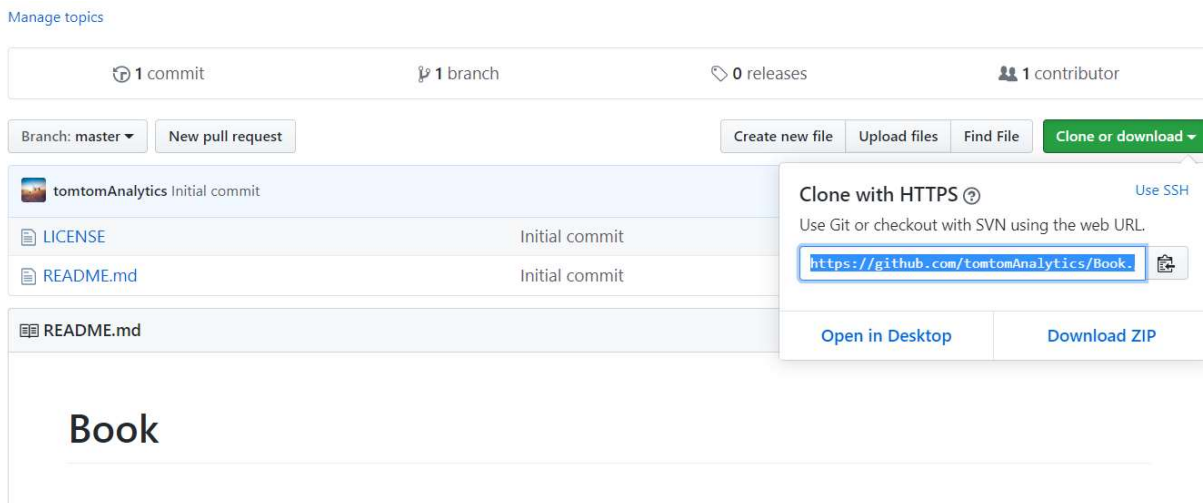
Create a Pull Request from your current branch
The current branch (gh-pages) is already published to GitHub. Create a pull request to propose and collaborate on your changes.
Branch menu or Ctrl R [Create Pull Request](#)

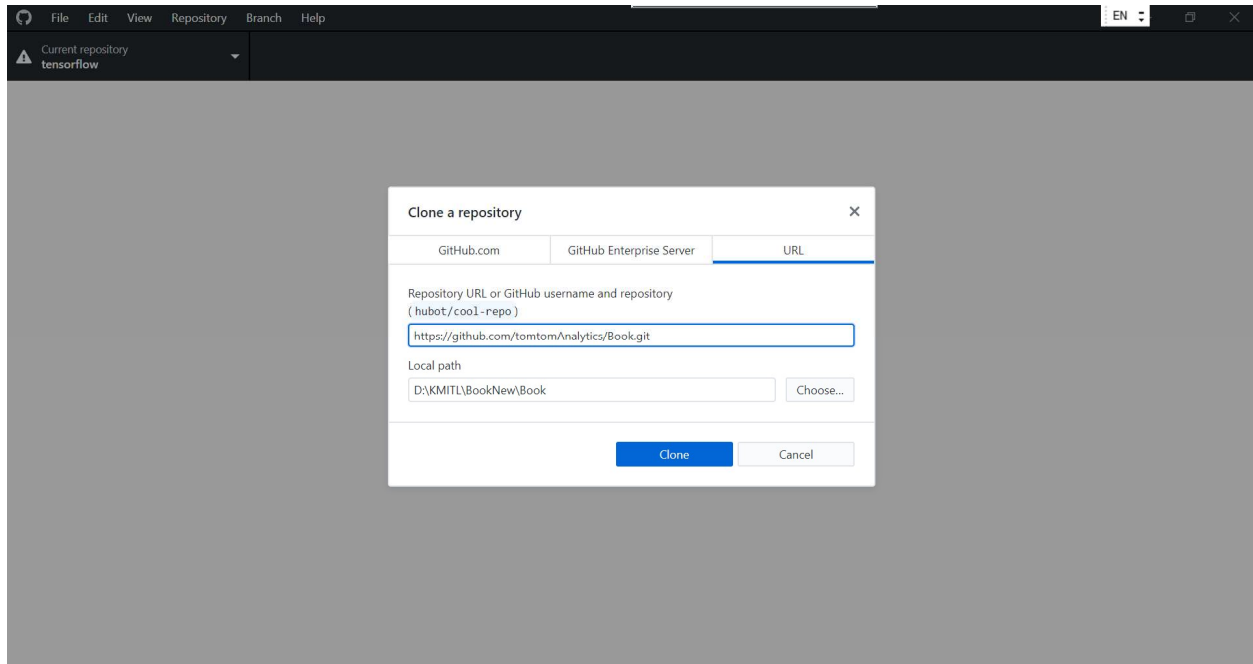
Open the repository in your external editor
Select your editor in [Options](#)
Repository menu or Ctrl Shift A [Open in Visual Studio Code](#)

View the files of your repository in Explorer [Show in Explorer](#)



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.

This PC > Data (D:) > KMITL > BookNew > Book

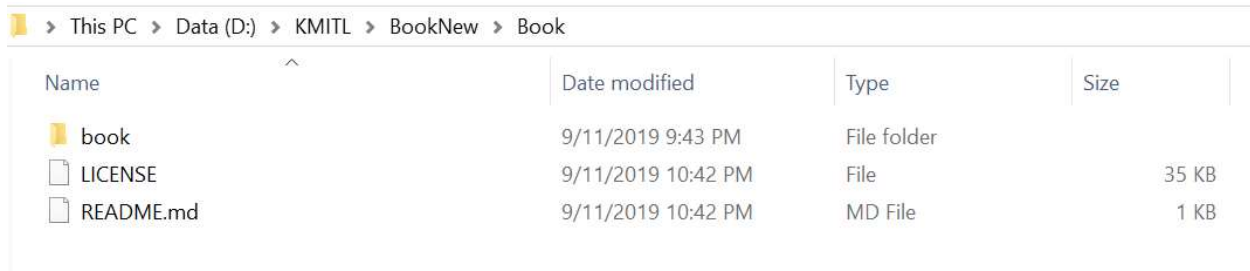
Name	Date modified	Type	Size
LICENSE	9/11/2019 10:42 PM	File	35 KB
README.md	9/11/2019 10:42 PM	MD File	1 KB

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.

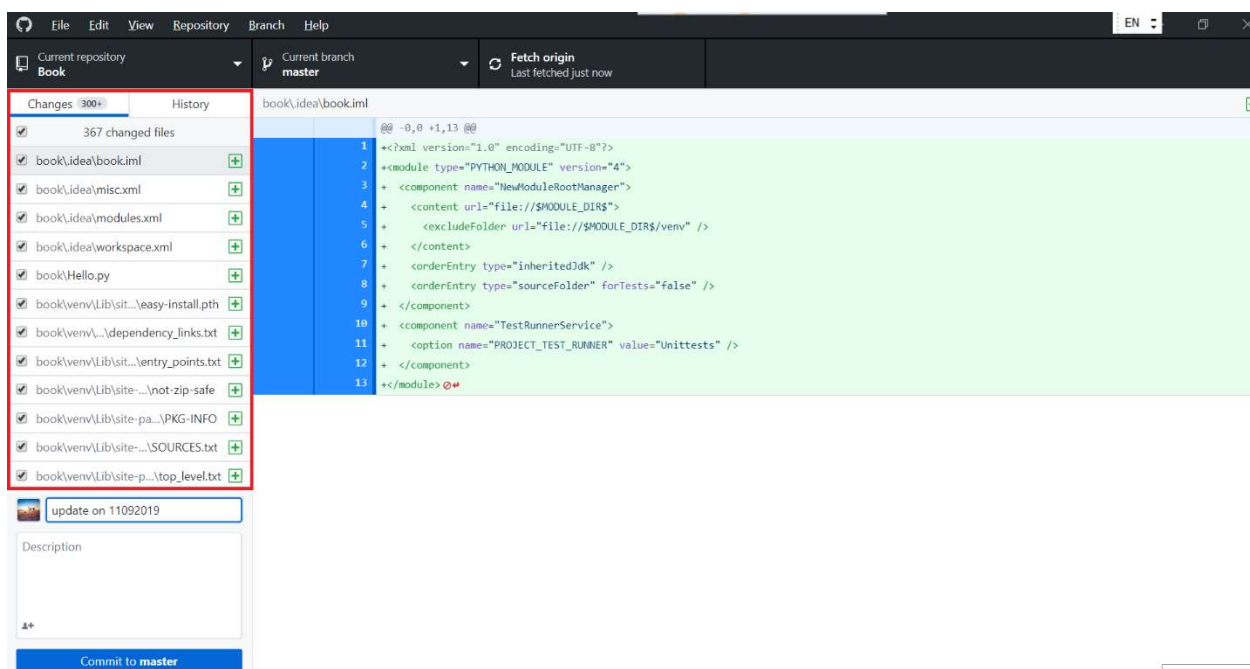


The screenshot shows a Windows File Explorer window with the address bar displaying the path: This PC > Data (D:) > KMITL > BookNew > Book. The main area shows a table of files and folders:

Name	Date modified	Type	Size
book	9/11/2019 9:43 PM	File folder	
LICENSE	9/11/2019 10:42 PM	File	35 KB
README.md	9/11/2019 10:42 PM	MD File	1 KB

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.

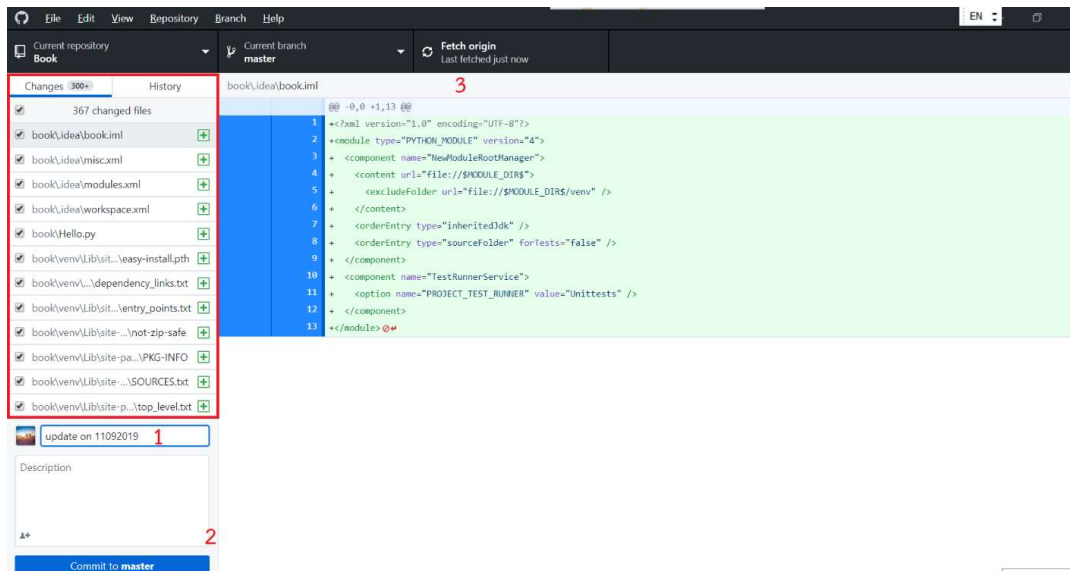


Committing Changes

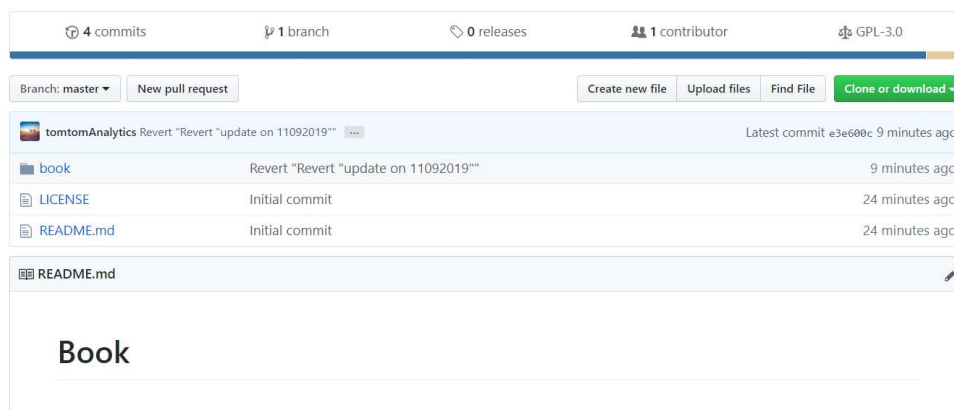
A commit tells Git that you made some changes which you want to record. Though a commit seems similar to saving a file, there are different aims behind ‘committing’ changes

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.

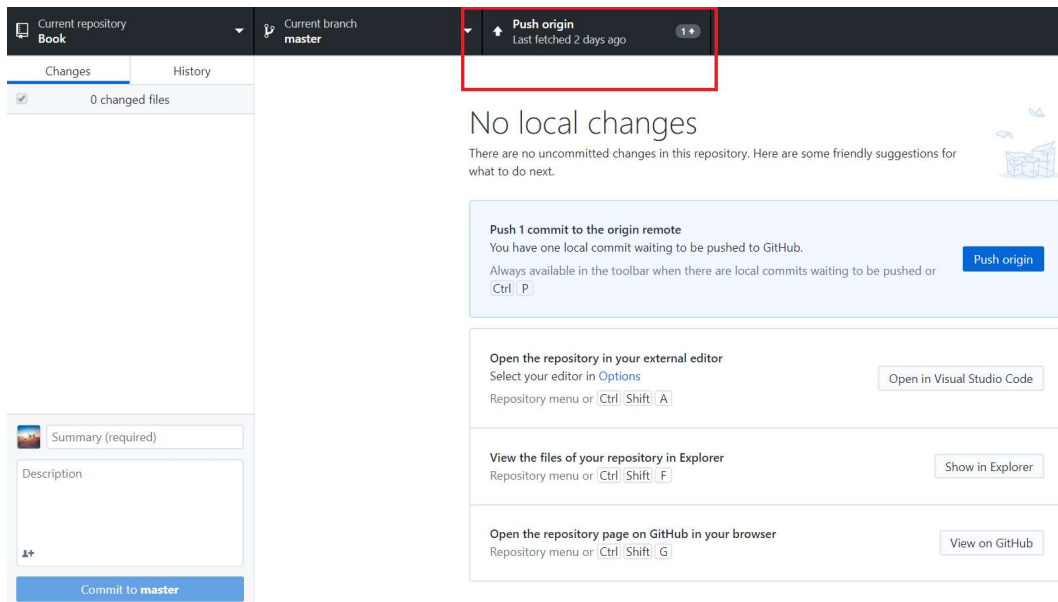
The method to commit a change is type some note in the box Summary(required) and press a Commit to master and following with Fetch origin to push all change to GitHub account.



Result

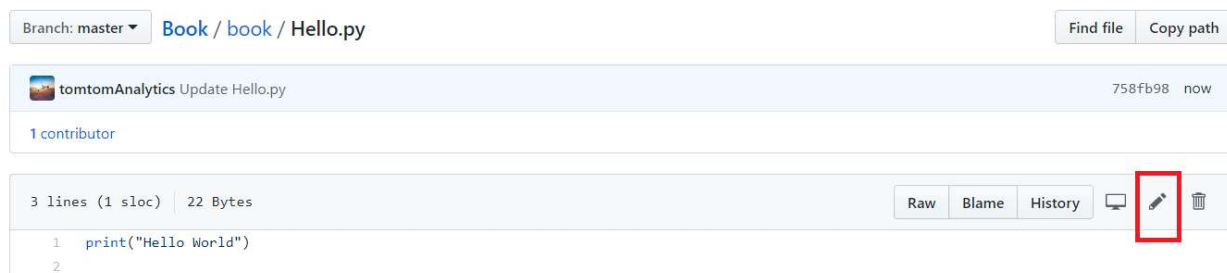


The next step is Push Origin to update the changing code in GitHub server.

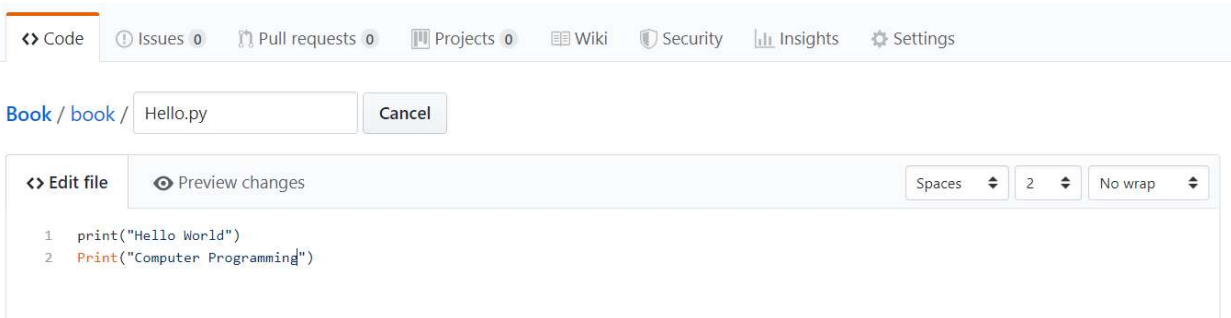


Making Changes Remotely

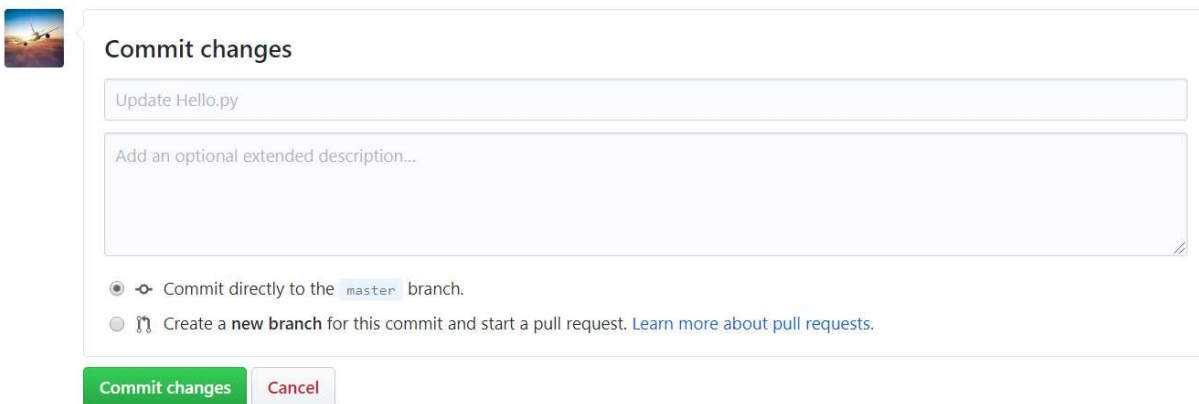
It is also possible to make a change to your repository on the web interface. Clicking on the name of the file will take you to a new page showing your document. From the web interface you have a variety of options available to you, including viewing the history of changes, viewing the file in GitHub 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.



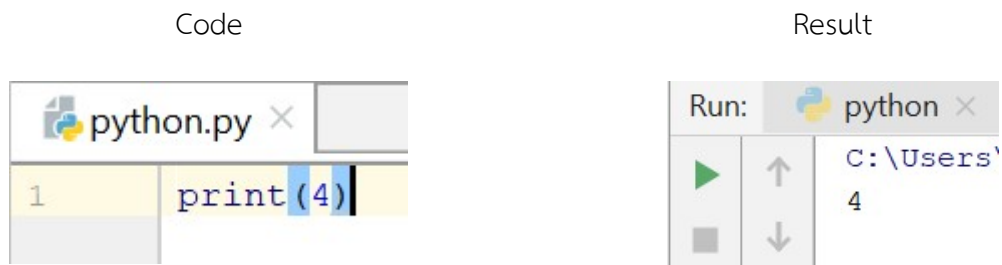
Once you have committed these changes they will be stored on the remote repository.

To get them back onto our computer we need to sync our these changes. We will see the 'Push Origin' button on GitHub Desktop.

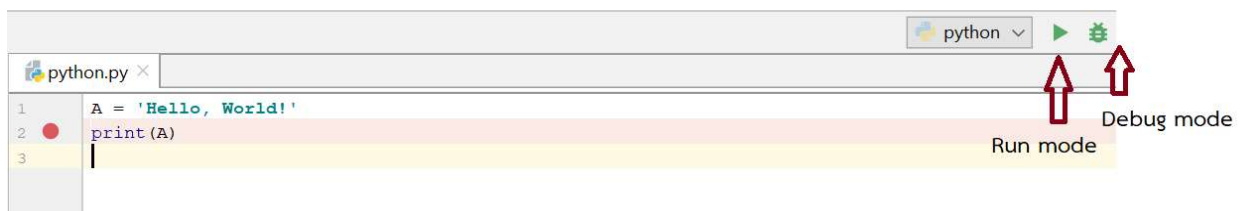
Chapter 2 Variables, expressions and statements

2.1 Values and types

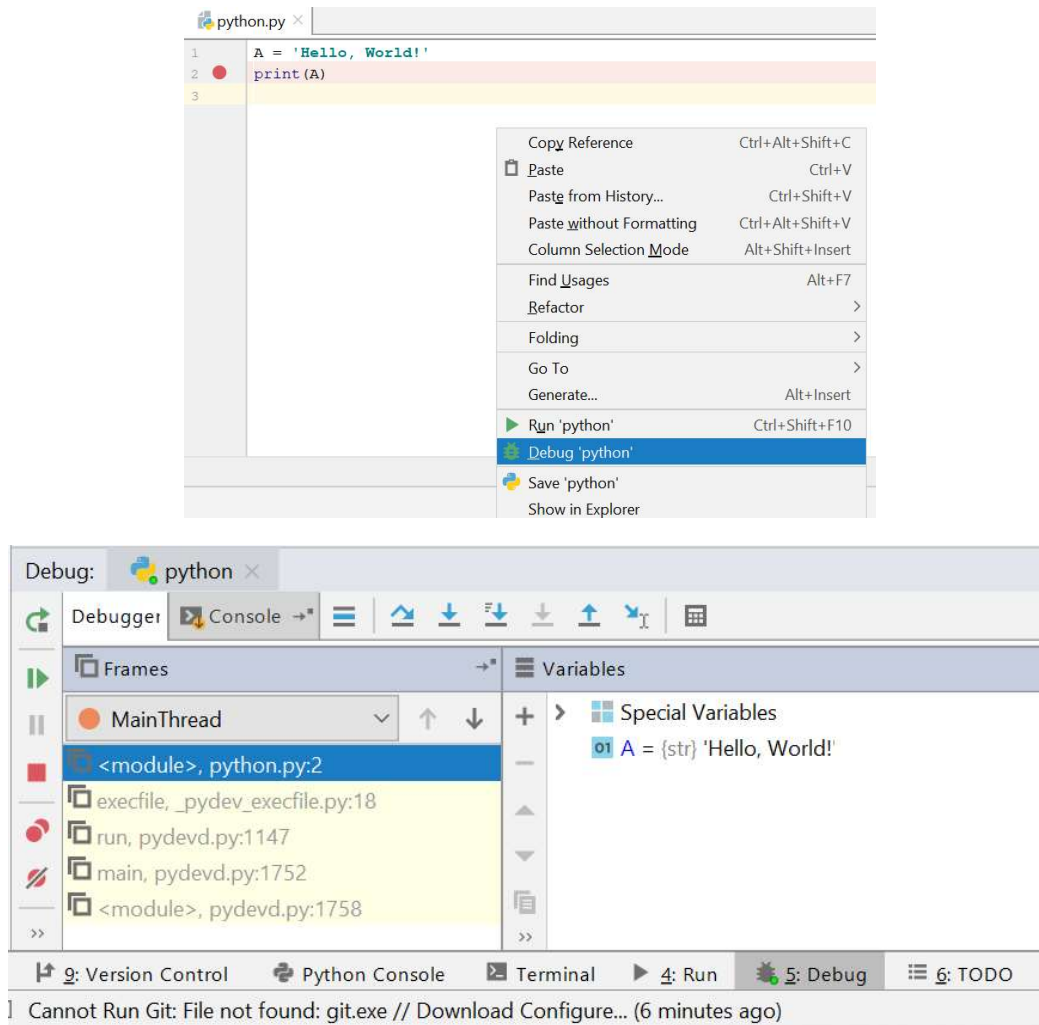
A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'. These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a “string” of letters. You (and the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



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

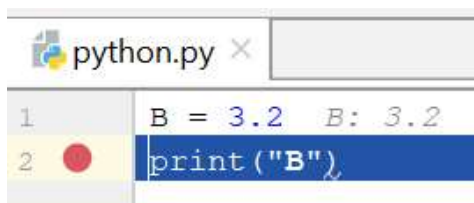


Or right click in coding area and then select debug mode. The result will show in result area as the following figure.

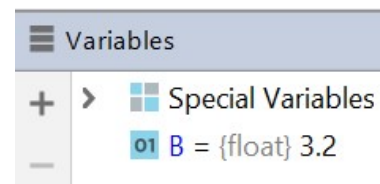


Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called floating-point.

Code



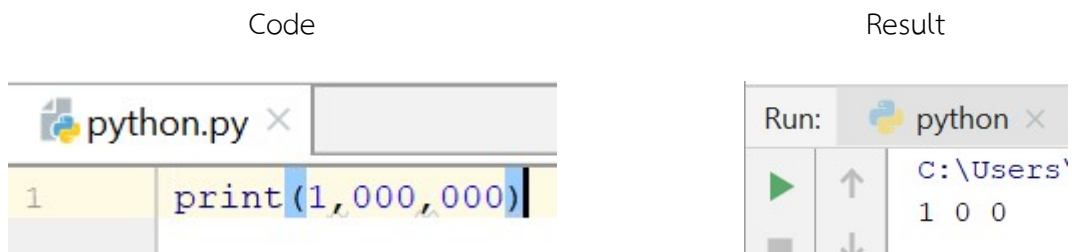
Result



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



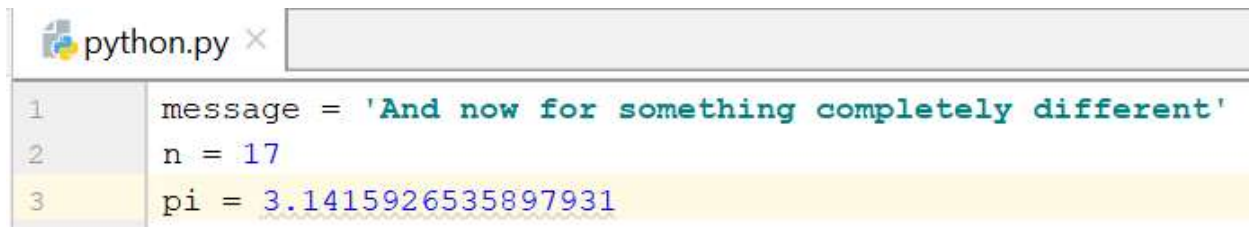
They're strings. When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:



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.

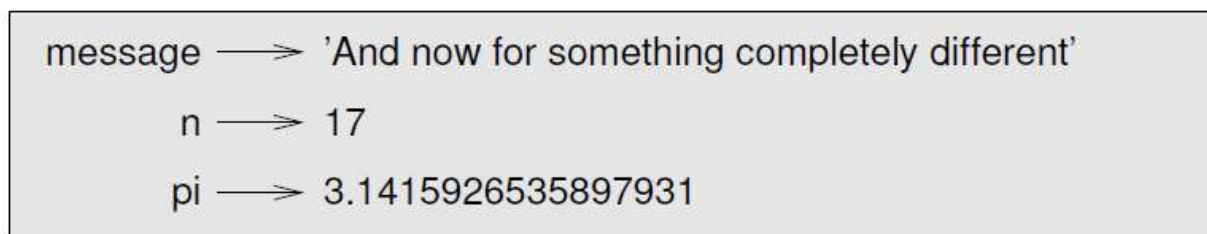
2.2 Variables

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:



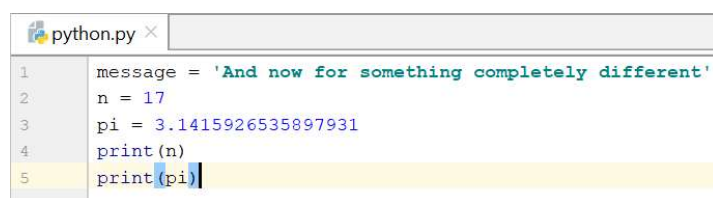
```
python.py x
1 message = 'And now for something completely different'
2 n = 17
3 pi = 3.1415926535897931
```

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:



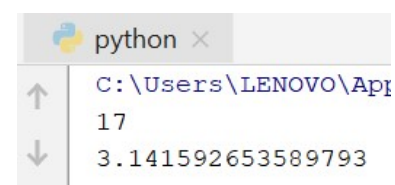
To display the value of a variable, you can use a print statement:

Code



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

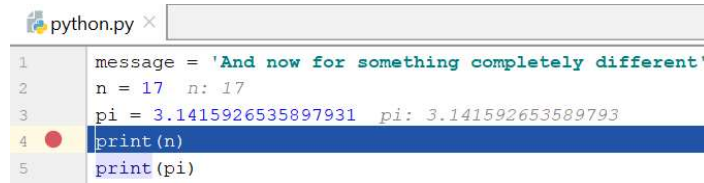
Result



```
python x
C:\Users\LENOVO\AppData\Local\Microsoft\Windows\In
17
3.141592653589793
```

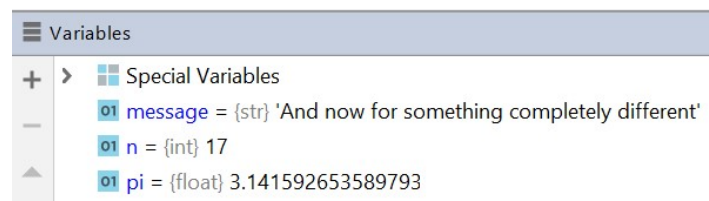
The type of a variable is the type of the value it refers to.

Code



```
python.py x
1 message = 'And now for something completely different'
2 n = 17 n: 17
3 pi = 3.1415926535897931 pi: 3.141592653589793
4 print(n)
5 print(pi)
```

Result

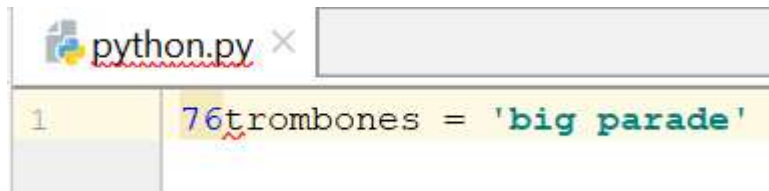


```
Variables
+ > Special Variables
- 01 message = {str} 'And now for something completely different'
  01 n = {int} 17
  01 pi = {float} 3.141592653589793
```

2.3 Variable names and keywords

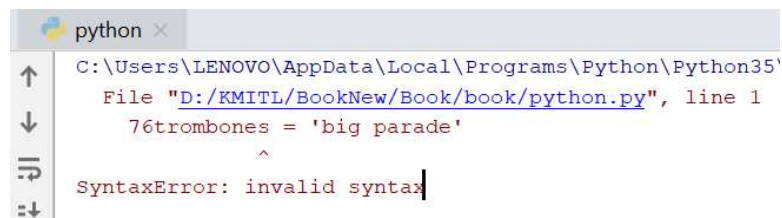
Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea 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



```
python.py x
1 76trombones = 'big parade'
```

Result



```
python x
C:\Users\LENOVO\AppData\Local\Programs\Python\Python35\
File "D:/KMITL/BookNew/Book/book/python.py", line 1
    76trombones = 'big parade'
        ^
SyntaxError: invalid syntax
```

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
```

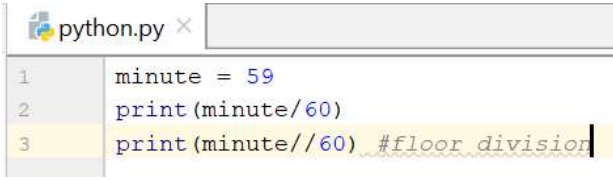
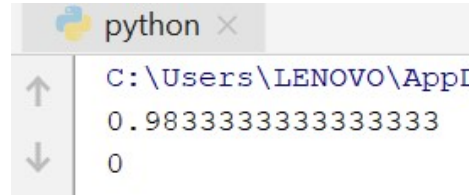
```
hour*60+minute
```

```
minute/60
```

```
5**2
(5+9) * (15-7)
```

In some other languages, \wedge 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
 <pre>python.py x 1 minute = 59 2 print(minute/60) 3 print(minute//60) #floor division</pre>	 <pre>python x C:\Users\LENOVO\AppData\Local\Microsoft\Windows\Apps\python.exe 0.9833333333333333 0</pre>

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):

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 $(\text{minute} * 100) / 60$, 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 $\text{degrees} / 2 * \text{pi}$, the division happens first and the result is multiplied by pi. To divide by 2pi , you can use parentheses or write $\text{degrees} / 2 / \text{pi}$.

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:

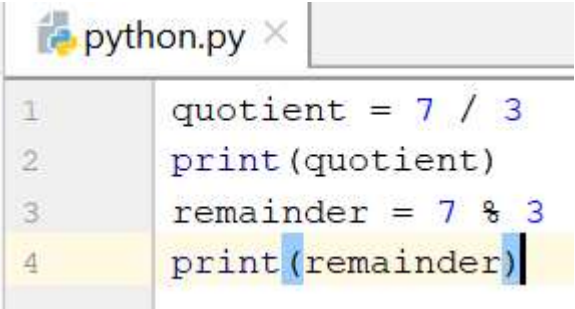
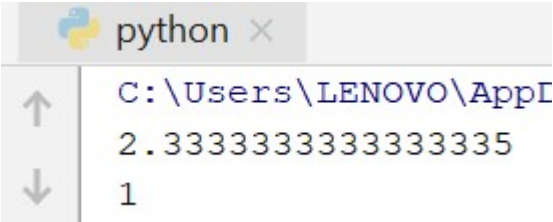
```
v = 5 # velocity in meters/second.
```

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
 <pre>python.py x 1 quotient = 7 / 3 2 print(quotient) 3 remainder = 7 % 3 4 print(remainder)</pre>	 <pre>python x C:\Users\LENOVO\AppData 2.3333333333333335 1</pre>

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
 <pre>python.py 1 print(5==5) 2 print(5==6)</pre>	 <pre>python C:\Users\l True False</pre>

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`

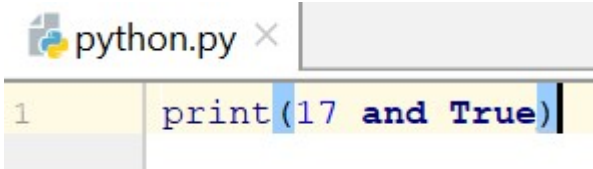

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a comparison operator.

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

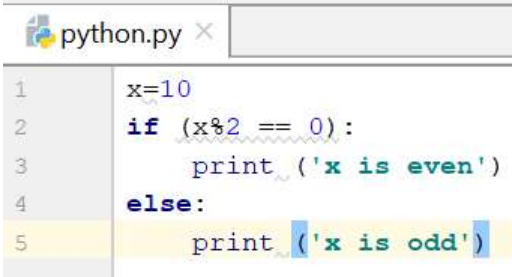
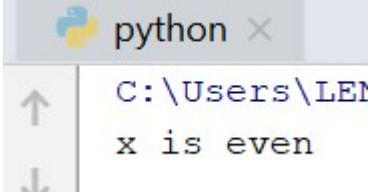
with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

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
 <pre>python.py x 1 x=10 2 if (x%2 == 0): 3 print('x is even') 4 else: 5 print('x is odd')</pre>	 <pre>python x C:\Users\LEI x is even</pre>

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

```
python.py x
1 x = 1
2 y = 5
3 if (x < y):
4     print('x is less than y')
5 elif (x > y):
6     print('x is greater than y')
7 else:
8     print('x and y are equal')
```

Result

```
python x
C:\Users\LENOVO\A
x is less than y
```

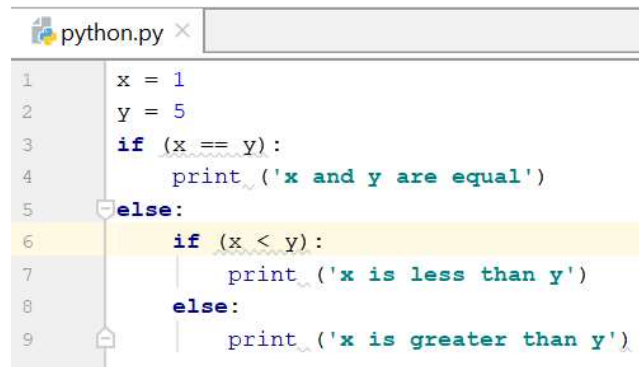
elif is an abbreviation of “else if.” Again, exactly one branch will be executed. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

```
python.py x
1 x = 1
2 y = 5
3 if (x < y):
4     print('x is less than y')
5 elif (x > y):
6     print('x is greater than y')
7 elif (x == y):
8     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:

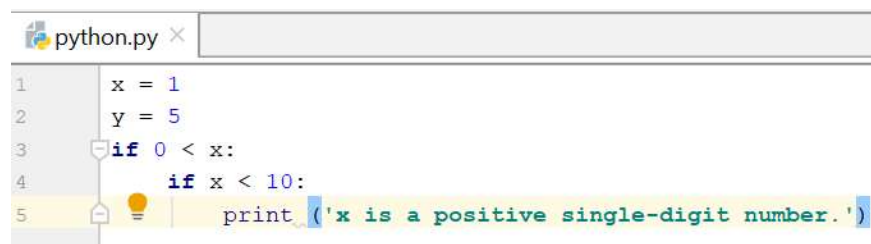


```
python.py x
1 x = 1
2 y = 5
3 if (x == y):
4     print('x and y are equal')
5 else:
6     if (x < y):
7         print('x is less than y')
8     else:
9         print('x is greater than y')
```

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.

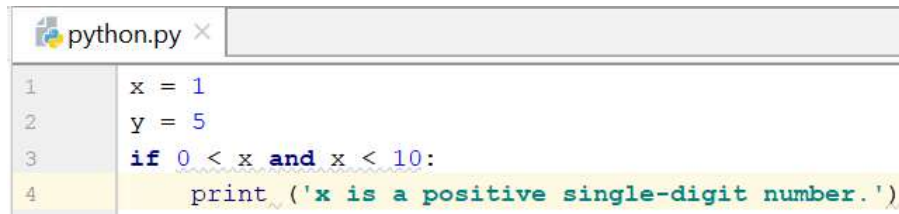
Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

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 x = 1
2 y = 5
3 if 0 < x:
4     if x < 10:
5         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:



```
python.py x
1 x = 1
2 y = 5
3 if 0 < x and x < 10:
4     print('x is a positive single-digit number.')
```

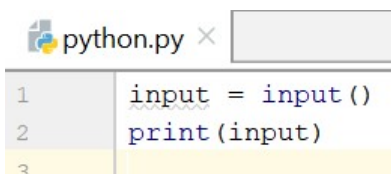
3.8 Keyboard input

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.

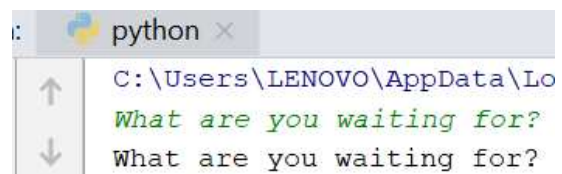
When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

Code



```
python.py x
1 input = input()
2 print(input)
3
```

Result



```
python x
C:\Users\LENOVO\AppData\Lo
What are you waiting for?
What are you waiting for?
```

Chapter 4 Loops and Iteration

There are two loop methods in this chapter namely For loop and While loop. The basic For loop is used when one knows the number of iterations and use While loop when one doesn't know the number of iterations a priori [7].

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

Result

```
python x
C:\Users\j
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

```
python.py x  
1 for x in "banana":  
2     print(x)
```

Result

```
python x  
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

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

Result

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

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

Code

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

Result

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

Result

```
python x
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

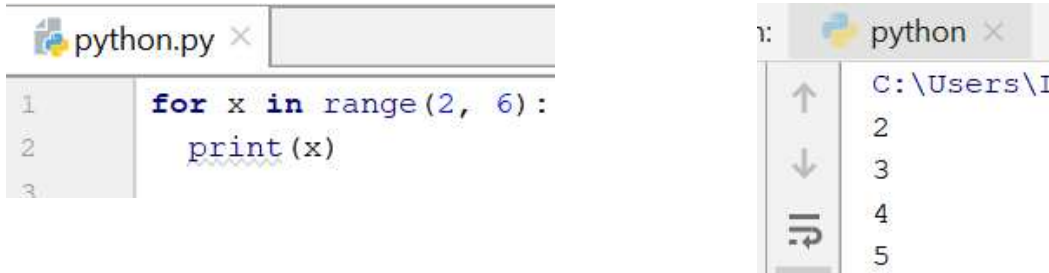
```
python.py x
1  for x in range(3):
2      print(x)
```

Result

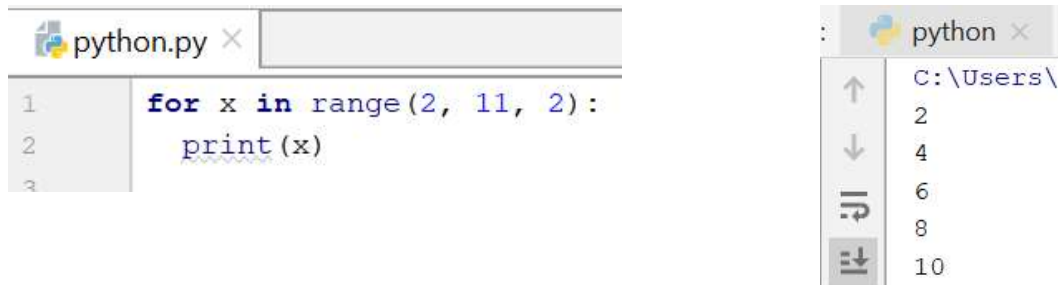
```
python x
C:\Users
0
1
2
```

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	Result
	

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	Result
	

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

```
python.py x
1 for x in range(3):
2     print(x)
3 else:
4     print("Finally finished!")
```

Result

```
python x
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 x
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

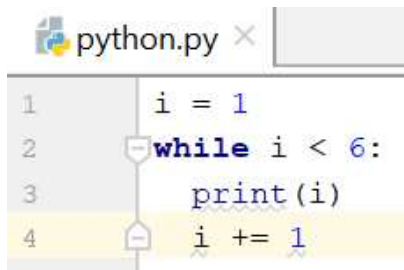
```
python x
C:\Users\LENOVO\
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

4.2 While loop

With the while loop we can execute a set of statements as long as a condition is true.

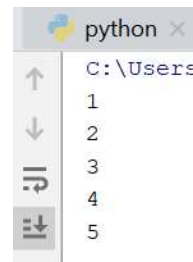
For example, print i as long as i is less than 6:

Code



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

Result



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

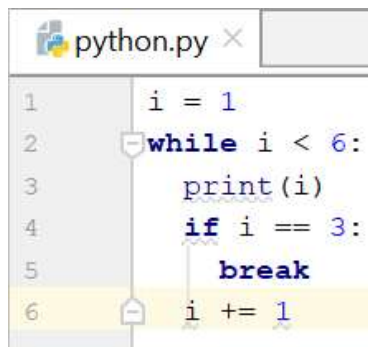
The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, *i*, which we set to 1.

4.2.1 The break Statement

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

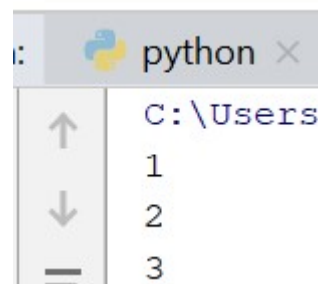
For example, exit the loop when *i* is 3:

Code



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

Result



```
python x
C:\Users
1
2
3
```

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

```
python.py x
1 i = 0
2 while i < 6:
3     i += 1
4     if i == 3:
5         continue
6     print(i)
```

Result

```
python x
C:\Users\
1
2
4
5
6
```

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

Chapter 5 List and Dictionaries

Chapter 6 Functions

Chapter 7 Files

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 Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



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	Sum of two variables
>>> x-2 3	Subtraction of two variables
>>> x*2 10	Multiplication of two variables
>>> x**2 25	Exponentiation of a variable
>>> x%2 1	Remainder of a variable
>>> x/float(2) 2.5	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]	Select item at index 1
>>> my_list[-3]	Select 3rd last item
Slice	
>>> my_list[1:3]	Select items at index 1 and 2
>>> my_list[1:]	Select items after index 0
>>> my_list[:3]	Select items before index 3
>>> my_list[:]	Copy my_list
Subset Lists of Lists	
>>> my_list2[1][0]	my_list[list][itemOfList]
>>> my_list2[1][:2]	

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
```



Install Python



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]	Select item at index 1
Slice	
>>> my_array[0:2]	Select items at index 0 and 1
array([1, 2])	
Subset 2D Numpy arrays	
>>> my_2darray[:,0]	my_2darray[rows, columns]
array([1, 4])	

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://desktop.github.com/>.
- [6] [Online]. Available: <https://programminghistorian.org/en/lessons/getting-started-with-github-desktop#what-are-git-and-github>.
- [7] [Online]. Available: https://www.w3schools.com/python/python_for_loops.asp.

Exercises & Solution