



# CPE252

## *COMPUTER PROGRAMMING FOR ENGINEERS*



Rong Phoophuangpairoj  
Department of Computer Engineering  
College of Engineering  
Rangsit University



# WHAT IS A PROGRAM?

A **program** is a sequence of instructions that specifies how to perform a computation.

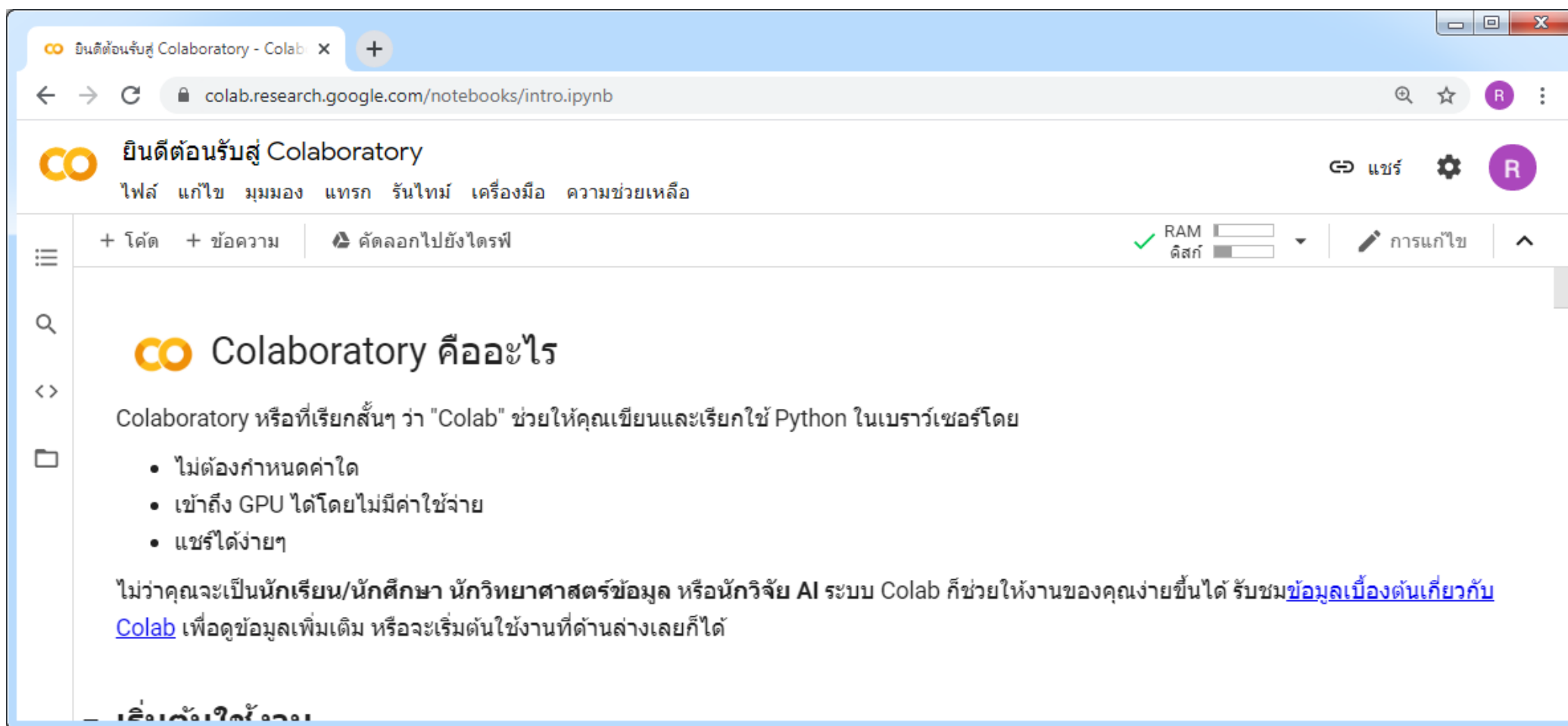
The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.



# RUNNING PYTHON

- One of the challenges of getting started with Python is that you might have to install Python and related software on your computer.
- There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it.

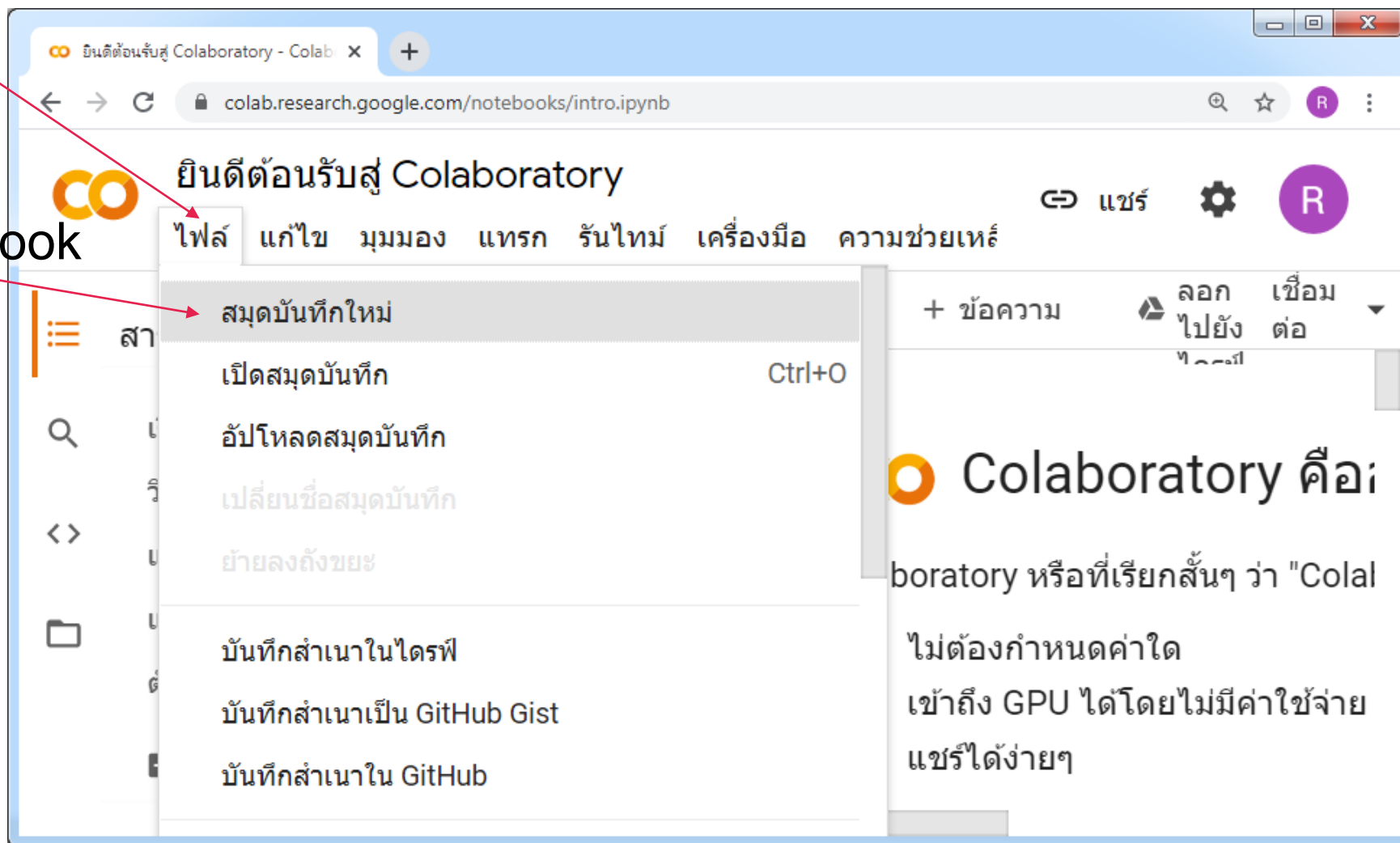
# HTTPS://COLAB.RESEARCH.GOOGLE.COM /NOTEBOOKS/INTRO.IPYNB (WRITE PROGRAMS ONLINE)



# NEW A NOTE BOOK (COLAB)

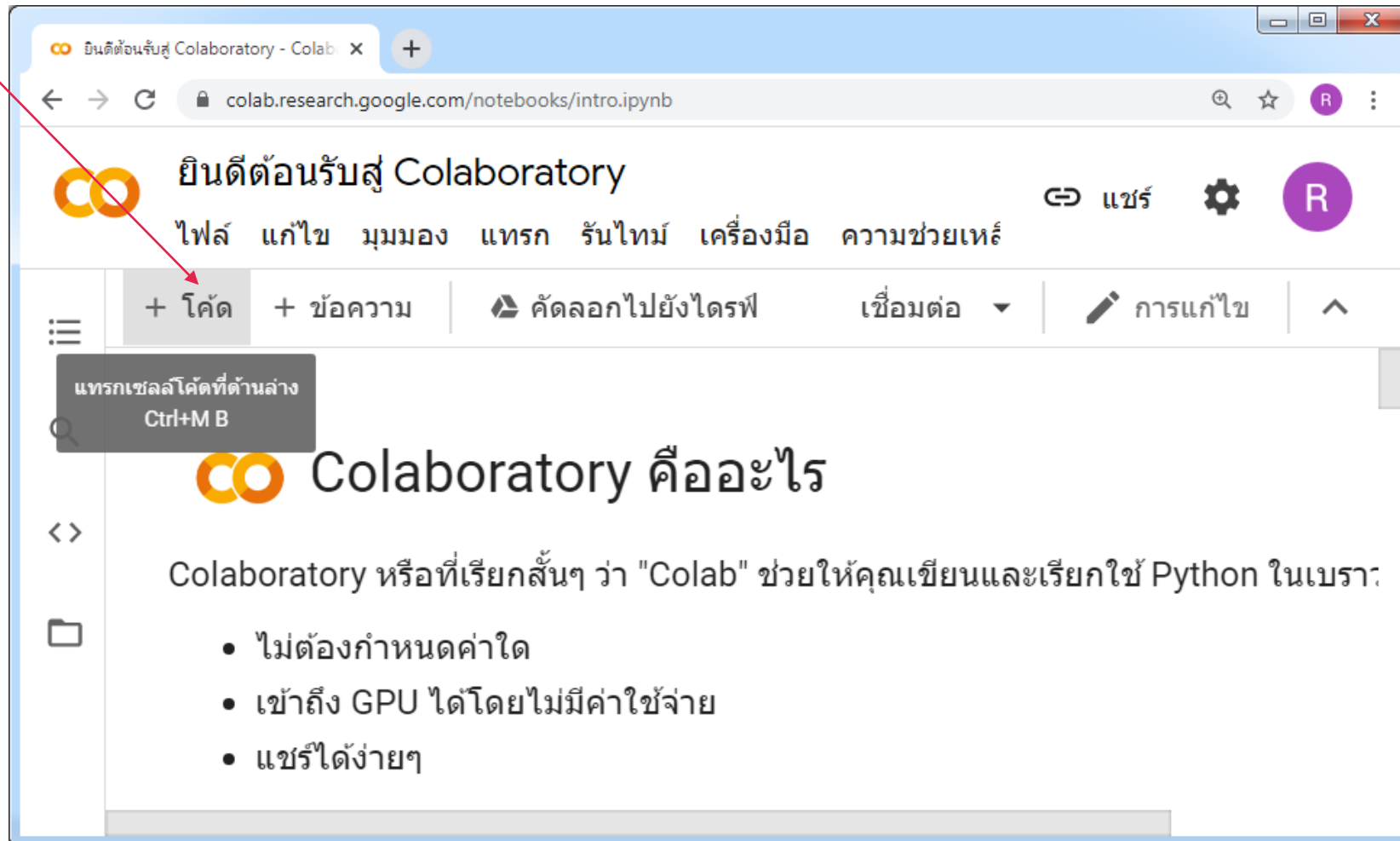
File

New a note book



# ADD A NEW PROGRAM

+ Code

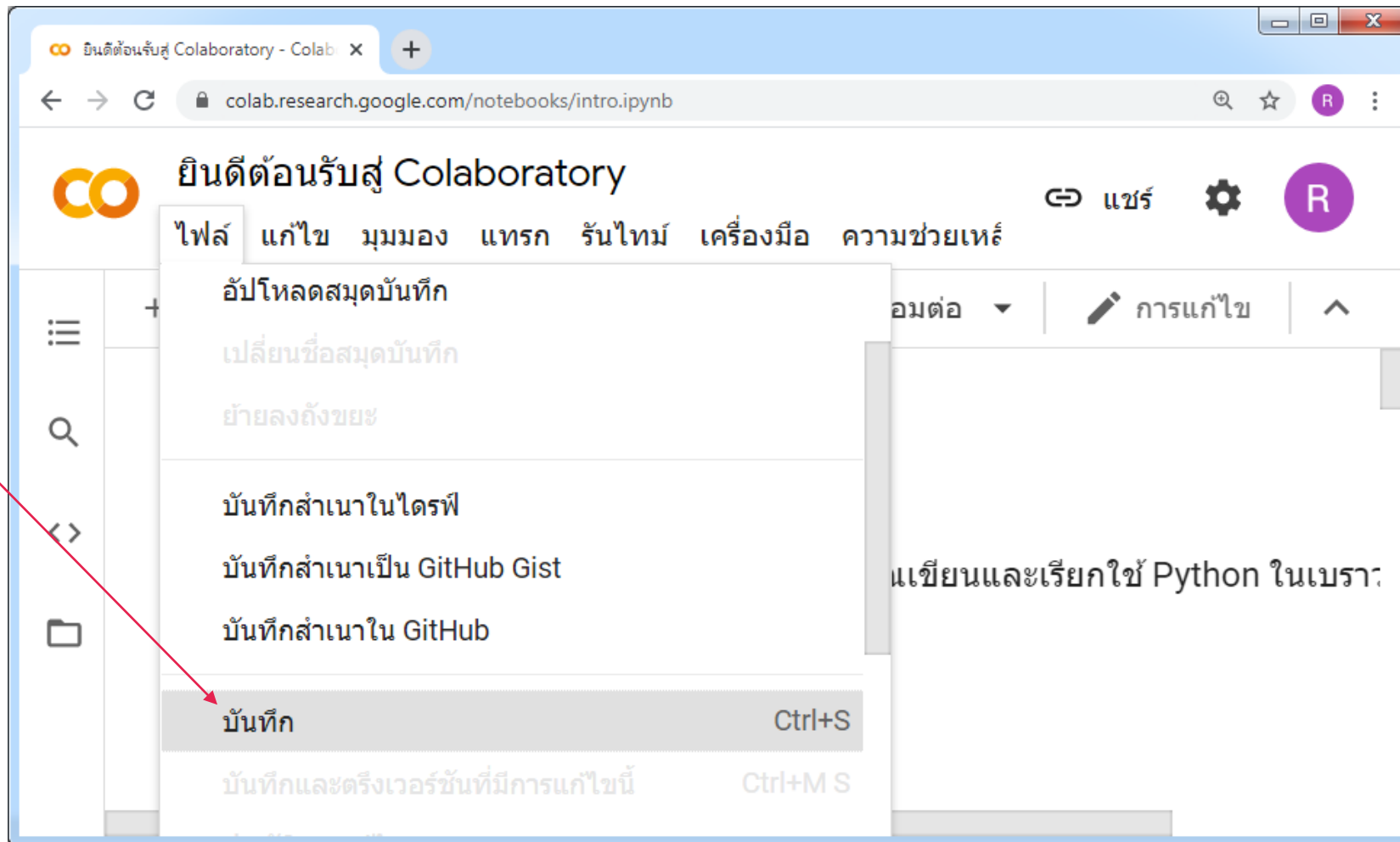


When using a smartphone, the menu may be different.

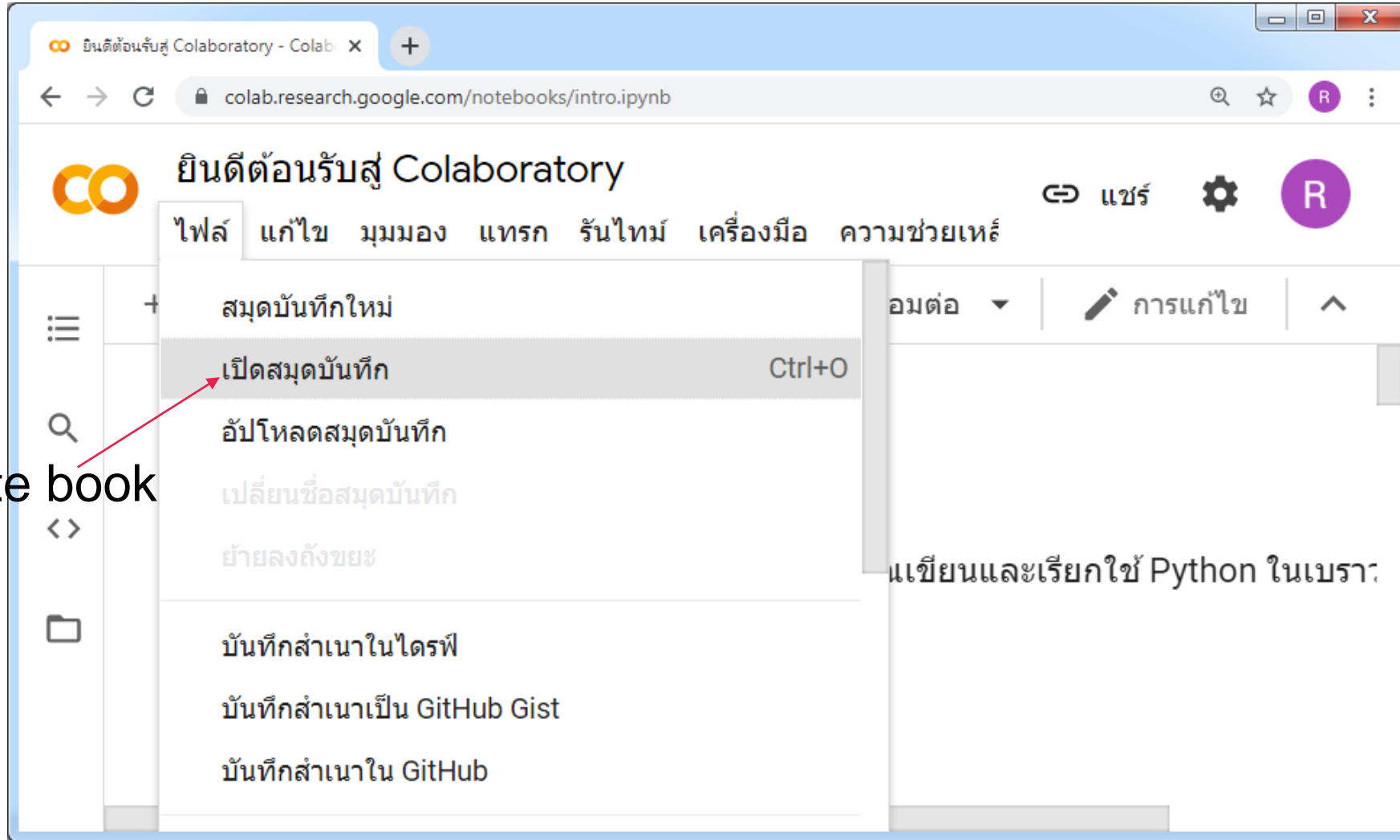


# SAVE

Save



# OPEN A NOTE BOOK



Open a note book

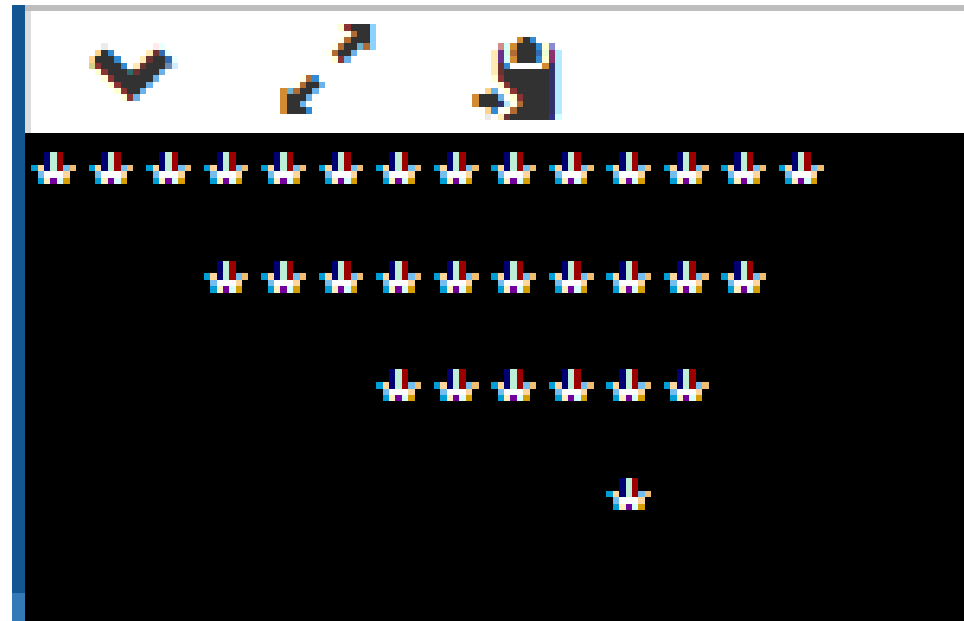


# ONLINEGDB.COM

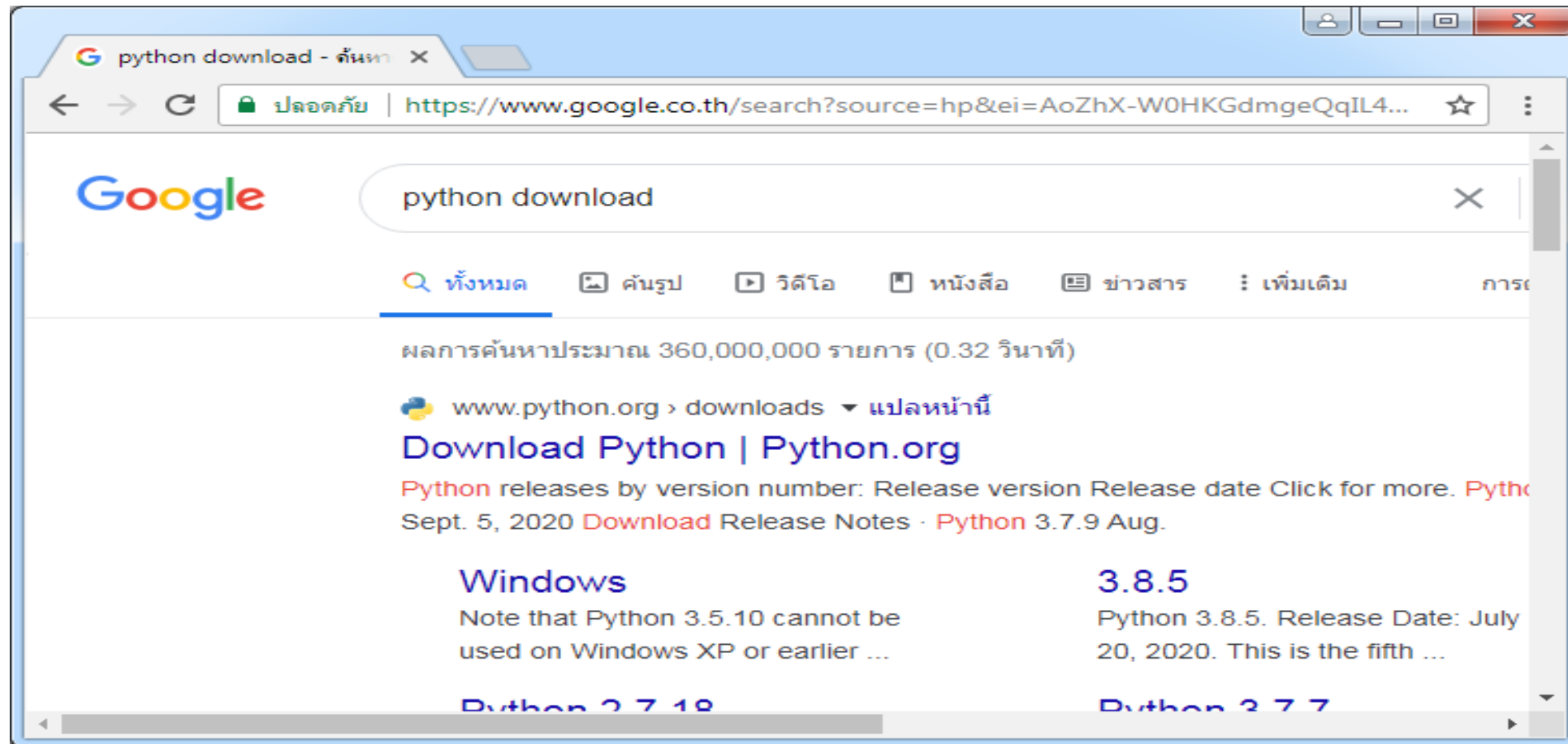
The screenshot displays the OnlineGDB website interface. The browser's address bar shows the URL `onlinegdb.com`. The page header includes the OnlineGDB logo, the text "online compiler and debugger for c/c++", and the tagline "code. compile. run. debug. share.". The main navigation menu on the left contains links for "IDE", "My Projects", "Classroom" (with a "new" badge), "Learn Programming", "Programming Questions", "Sign Up", and "Login". At the bottom of the menu are social media icons for Facebook and Twitter, along with a "+ 103K" badge. The top toolbar features buttons for "Run", "Debug", "Stop", "Share", "Save", "Beautify", and a download icon. The language is set to "Python 3". The code editor shows a file named `main.py` with the following Python code:

```
1 print()  
2 print('*****')  
3 print('*****')  
4 print('*****')  
5 print('*****')  
6
```

# OUTPUT



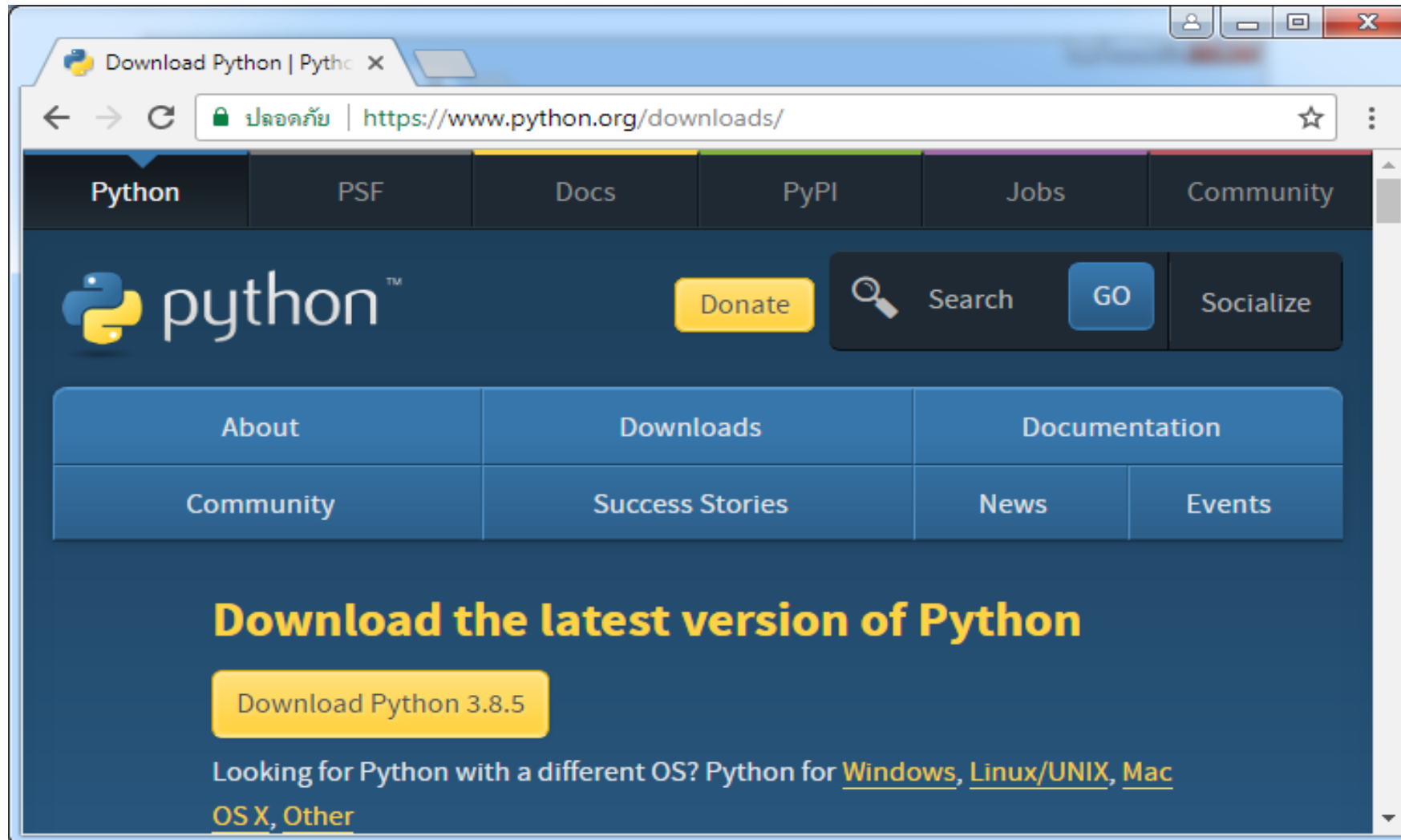
# DOWNLOAD PYTHON (WRITE PROGRAMS OFFLINE)



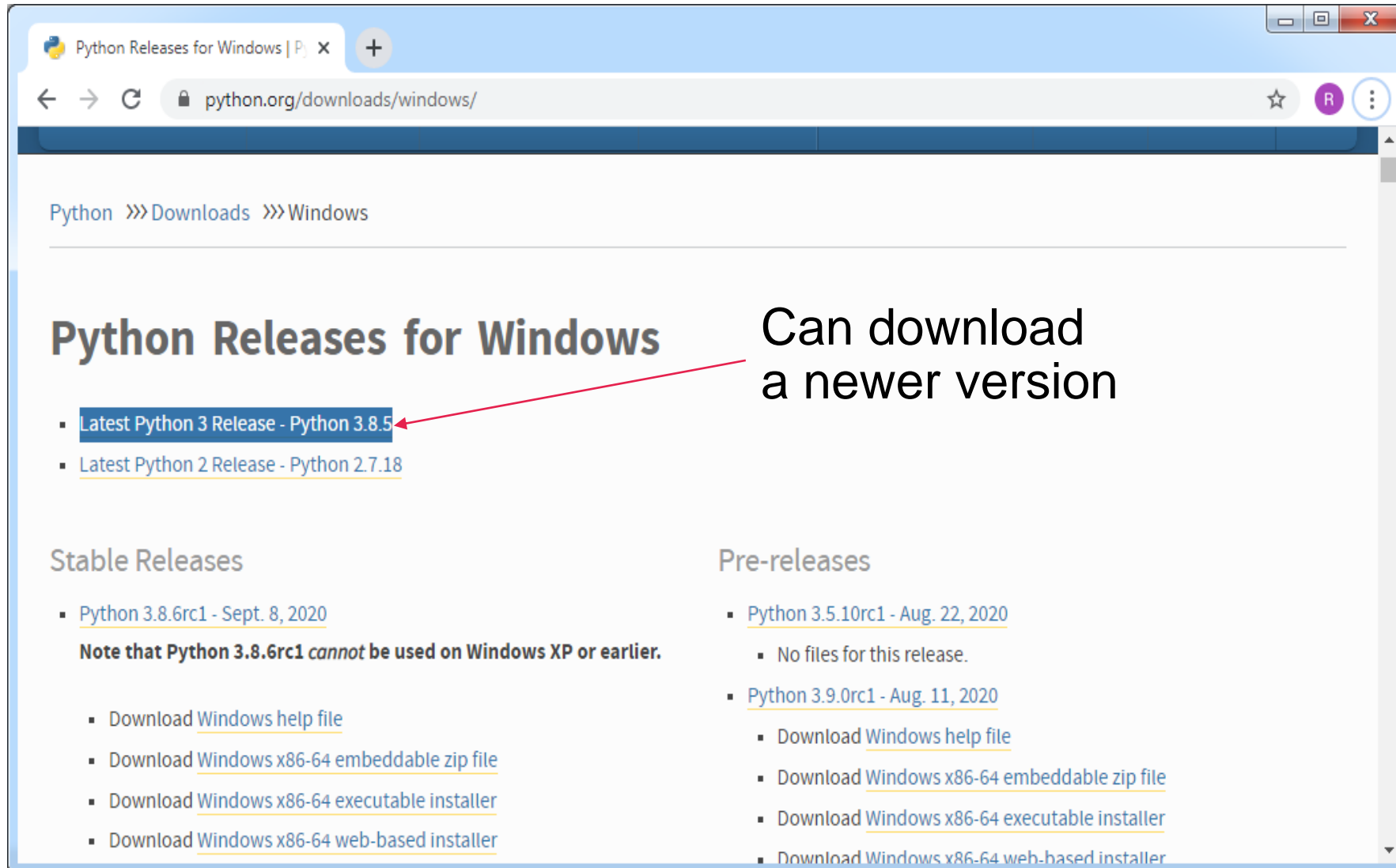
# INSTALLING A PYTHON IN YOUR COMPUTER



# DOWNLOAD PYTHON (CAN USE NEWER VERSION)



# DOWNLOAD PYTHON



The screenshot shows a web browser window with the address bar displaying `python.org/downloads/windows/`. The page title is "Python Releases for Windows". Below the title, there is a list of releases. A red arrow points from the text "Can download a newer version" to the link "Latest Python 3 Release - Python 3.8.5".

Python >>> Downloads >>> Windows

## Python Releases for Windows

- [Latest Python 3 Release - Python 3.8.5](#)
- [Latest Python 2 Release - Python 2.7.18](#)

Can download a newer version

### Stable Releases

- [Python 3.8.6rc1 - Sept. 8, 2020](#)  
**Note that Python 3.8.6rc1 cannot be used on Windows XP or earlier.**
  - Download [Windows help file](#)
  - Download [Windows x86-64 embeddable zip file](#)
  - Download [Windows x86-64 executable installer](#)
  - Download [Windows x86-64 web-based installer](#)

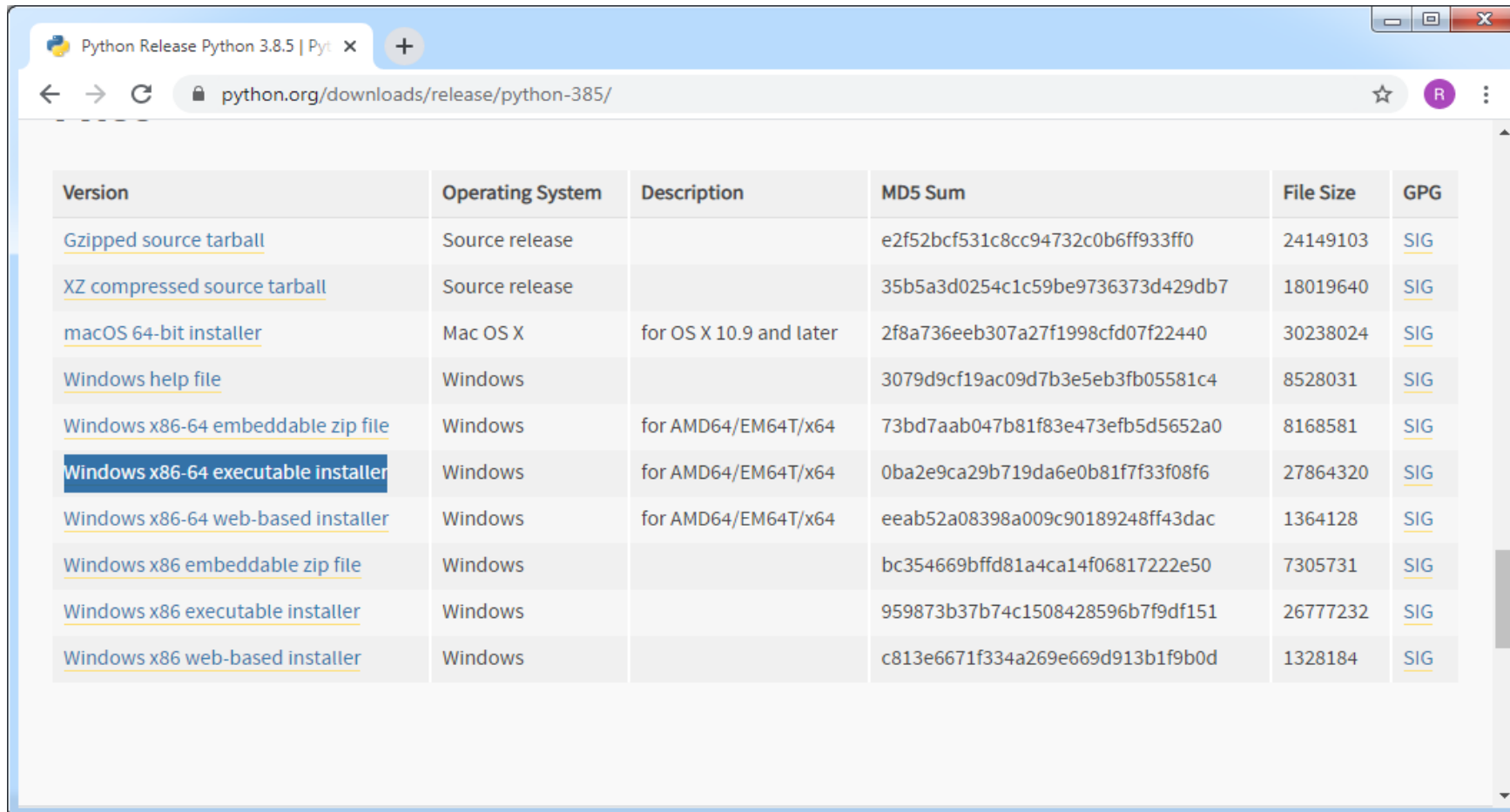
### Pre-releases

- [Python 3.5.10rc1 - Aug. 22, 2020](#)
  - No files for this release.
- [Python 3.9.0rc1 - Aug. 11, 2020](#)
  - Download [Windows help file](#)
  - Download [Windows x86-64 embeddable zip file](#)
  - Download [Windows x86-64 executable installer](#)
  - Download [Windows x86-64 web-based installer](#)



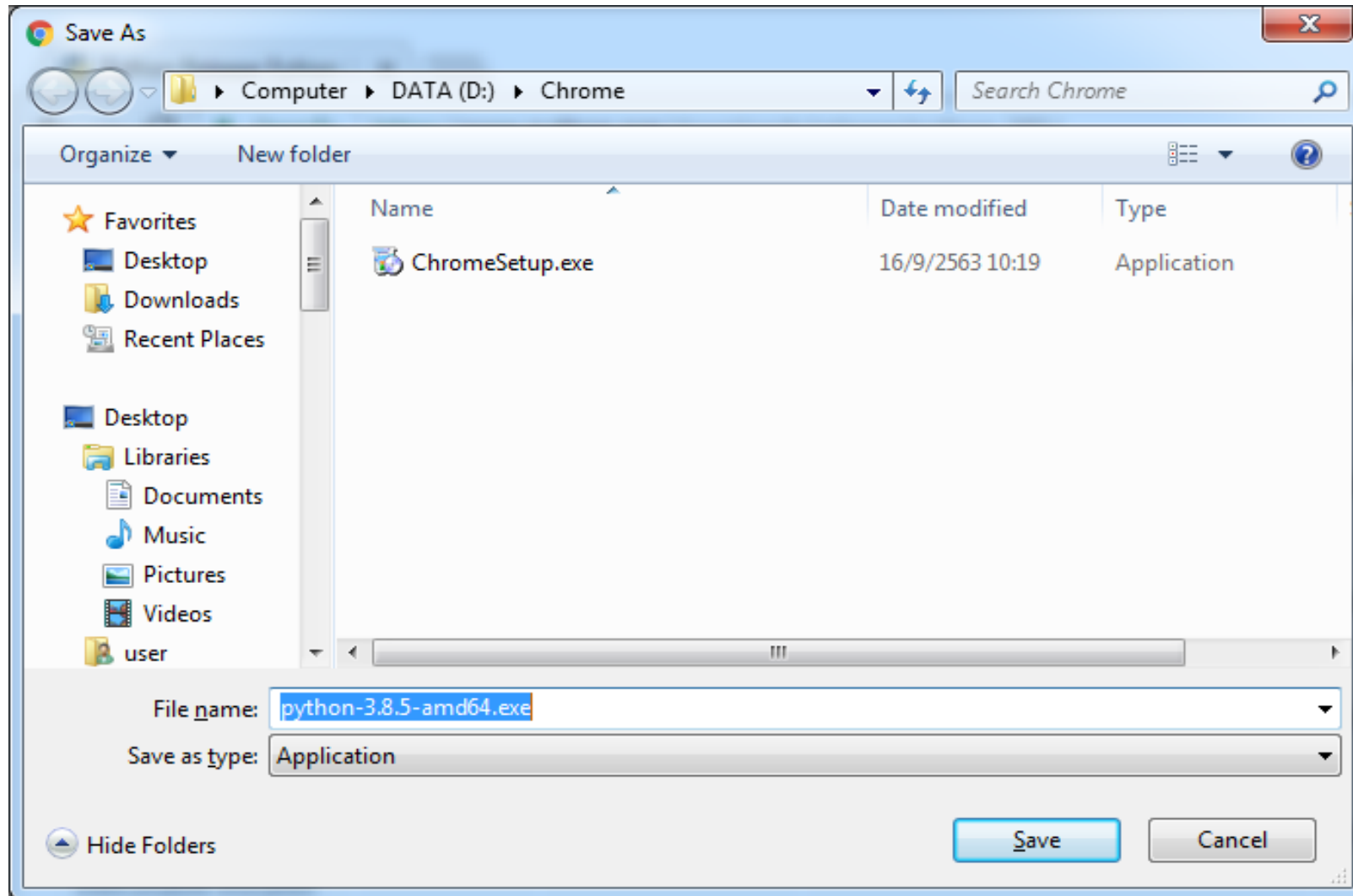
# DOWNLOAD PYTHON

Select a link to download either the **Windows x86-64 executable installer** or **Windows x86 executable installer**.

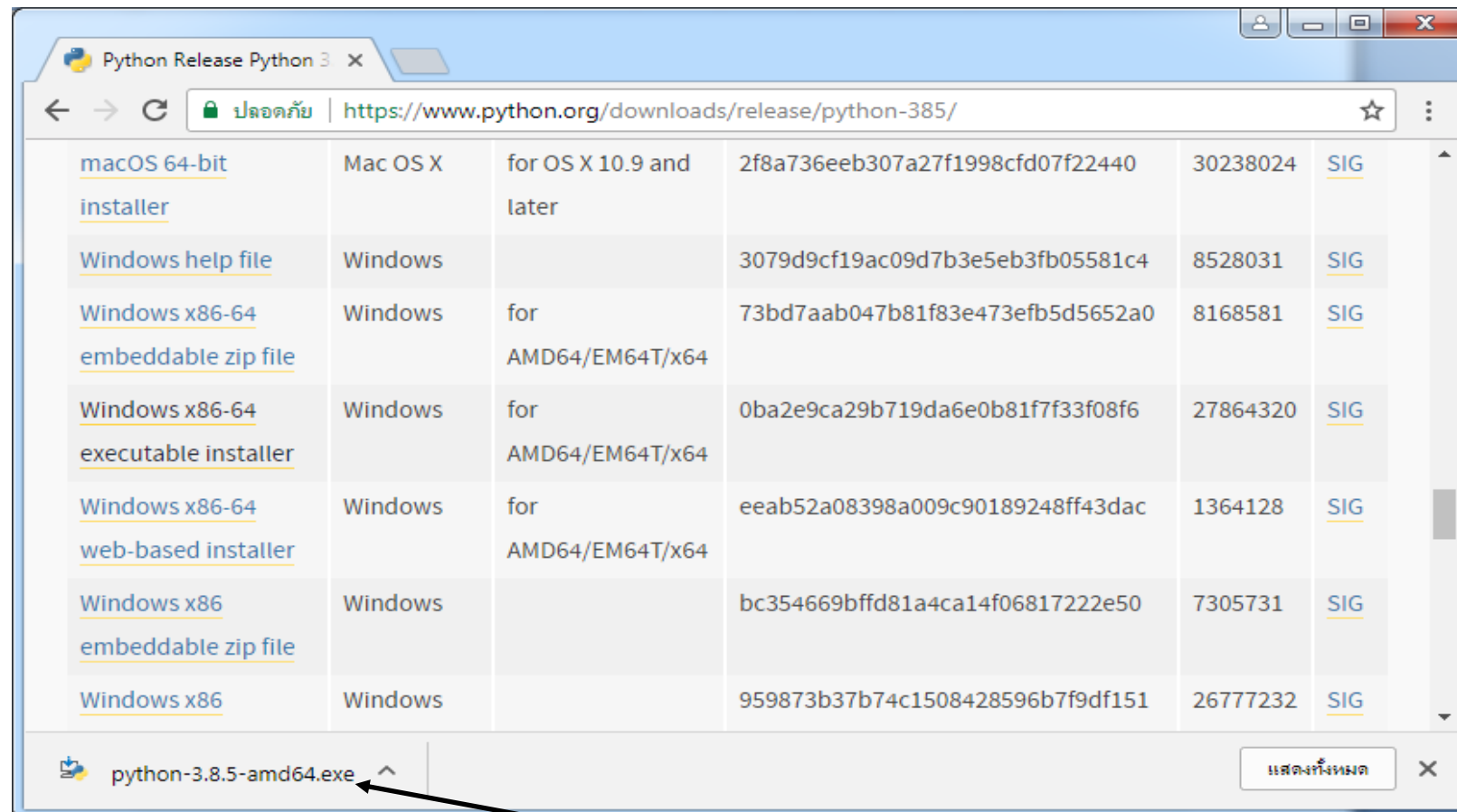
A screenshot of a web browser showing the Python 3.8.5 download page. The browser's address bar shows the URL 'python.org/downloads/release/python-385/'. The page contains a table with download links for various operating systems and architectures. The 'Windows x86-64 executable installer' link is highlighted with a blue background.

Version	Operating System	Description	MD5 Sum	File Size	GPG
<a href="#">Gzipped source tarball</a>	Source release		e2f52bcf531c8cc94732c0b6ff933ff0	24149103	<a href="#">SIG</a>
<a href="#">XZ compressed source tarball</a>	Source release		35b5a3d0254c1c59be9736373d429db7	18019640	<a href="#">SIG</a>
<a href="#">macOS 64-bit installer</a>	Mac OS X	for OS X 10.9 and later	2f8a736eeb307a27f1998cfd07f22440	30238024	<a href="#">SIG</a>
<a href="#">Windows help file</a>	Windows		3079d9cf19ac09d7b3e5eb3fb05581c4	8528031	<a href="#">SIG</a>
<a href="#">Windows x86-64 embeddable zip file</a>	Windows	for AMD64/EM64T/x64	73bd7aab047b81f83e473efb5d5652a0	8168581	<a href="#">SIG</a>
<b><a href="#">Windows x86-64 executable installer</a></b>	Windows	for AMD64/EM64T/x64	0ba2e9ca29b719da6e0b81f7f33f08f6	27864320	<a href="#">SIG</a>
<a href="#">Windows x86-64 web-based installer</a>	Windows	for AMD64/EM64T/x64	eeab52a08398a009c90189248ff43dac	1364128	<a href="#">SIG</a>
<a href="#">Windows x86 embeddable zip file</a>	Windows		bc354669bffd81a4ca14f06817222e50	7305731	<a href="#">SIG</a>
<a href="#">Windows x86 executable installer</a>	Windows		959873b37b74c1508428596b7f9df151	26777232	<a href="#">SIG</a>
<a href="#">Windows x86 web-based installer</a>	Windows		c813e6671f334a269e669d913b1f9b0d	1328184	<a href="#">SIG</a>

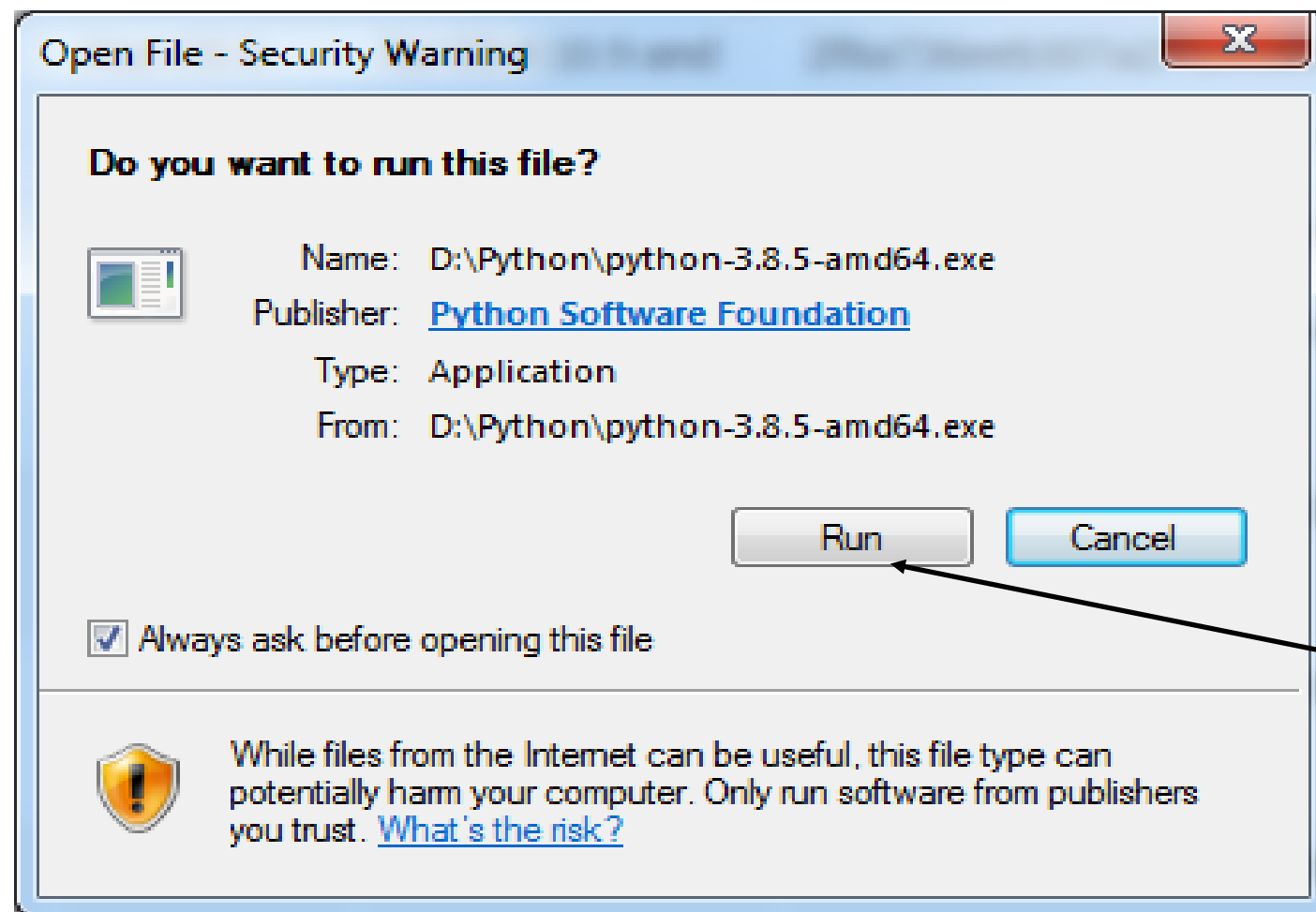
# DOWNLOAD PYTHON



# RUN EXECUTABLE INSTALLER

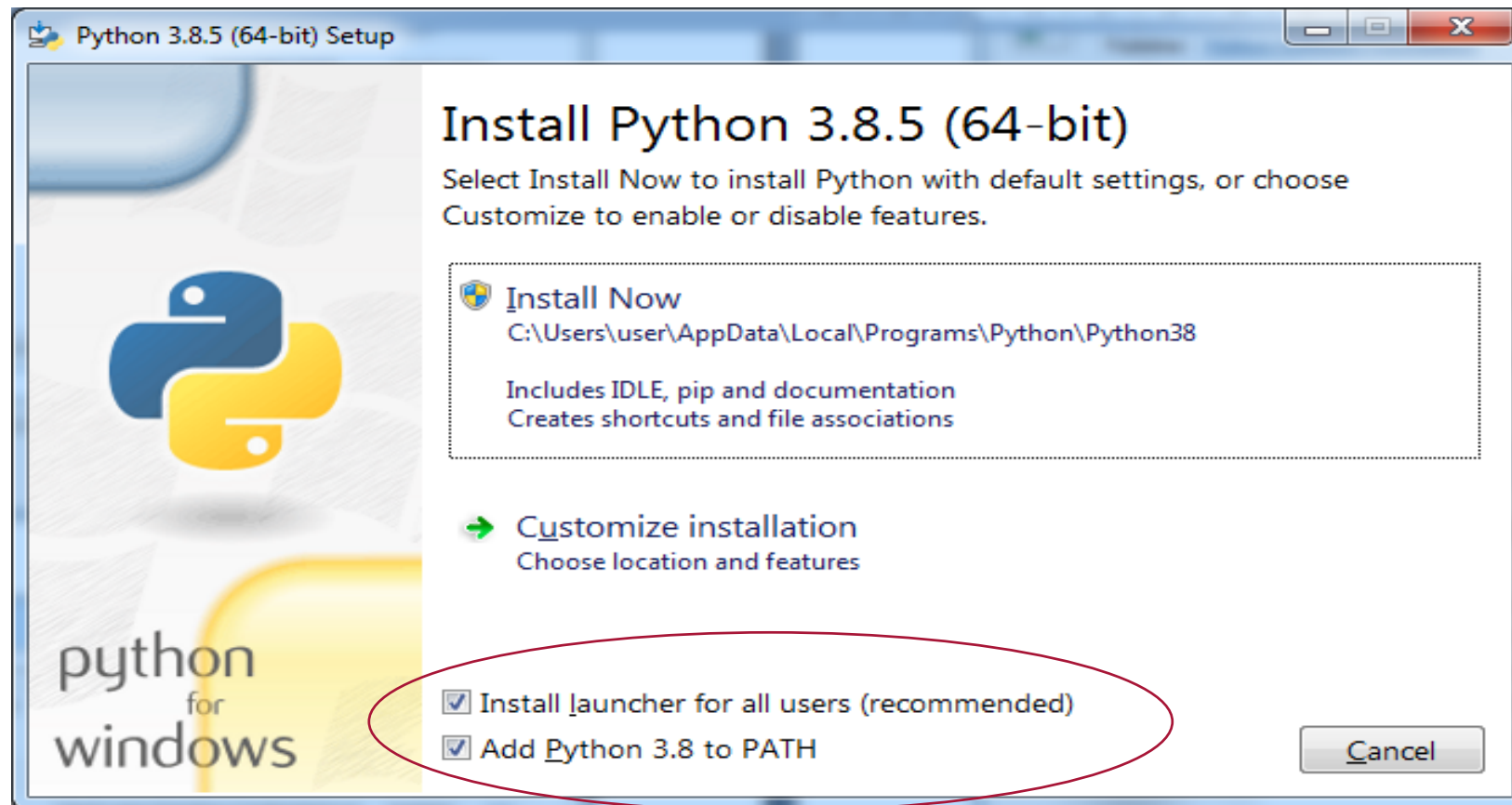


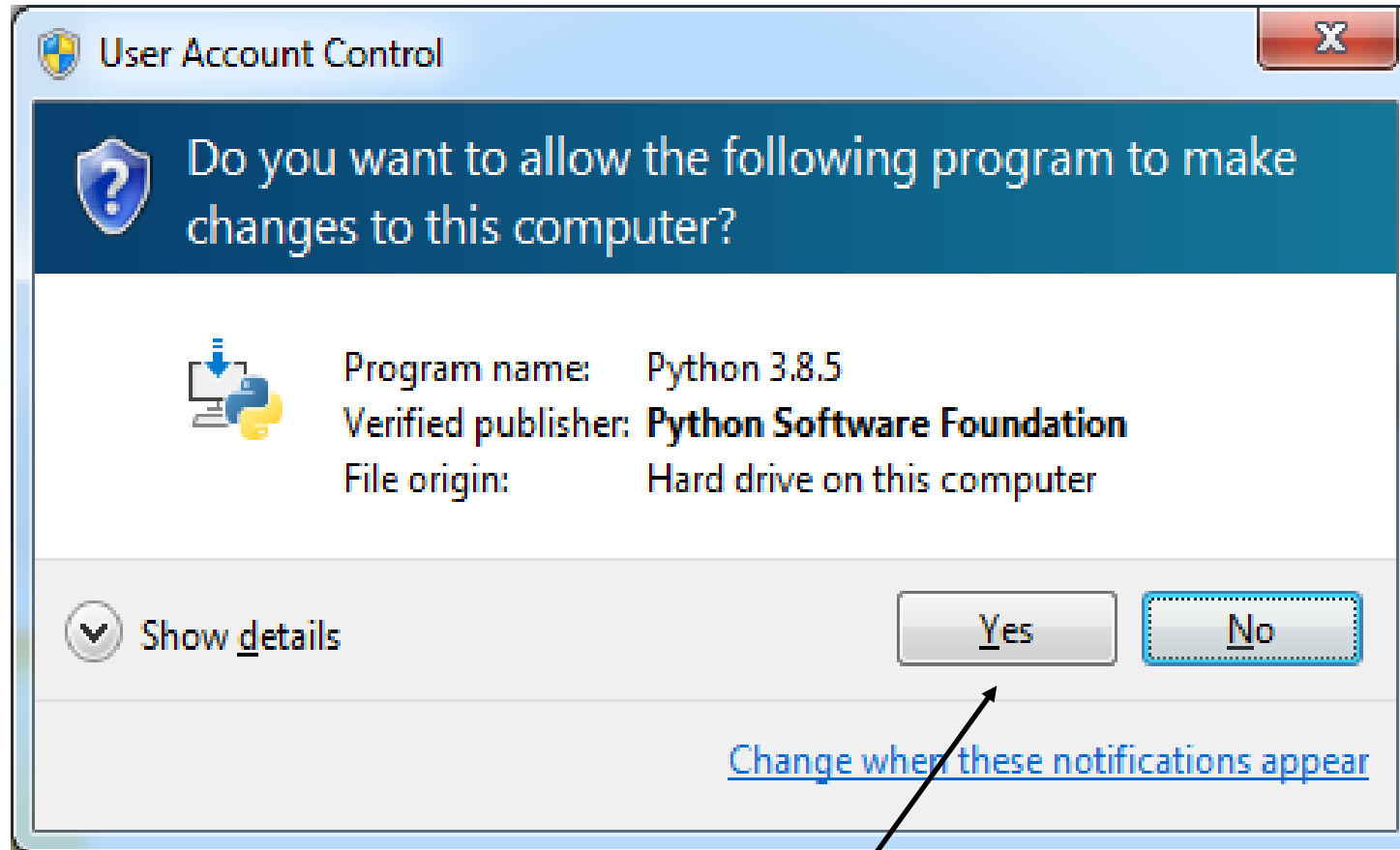
double click



Click on Run button

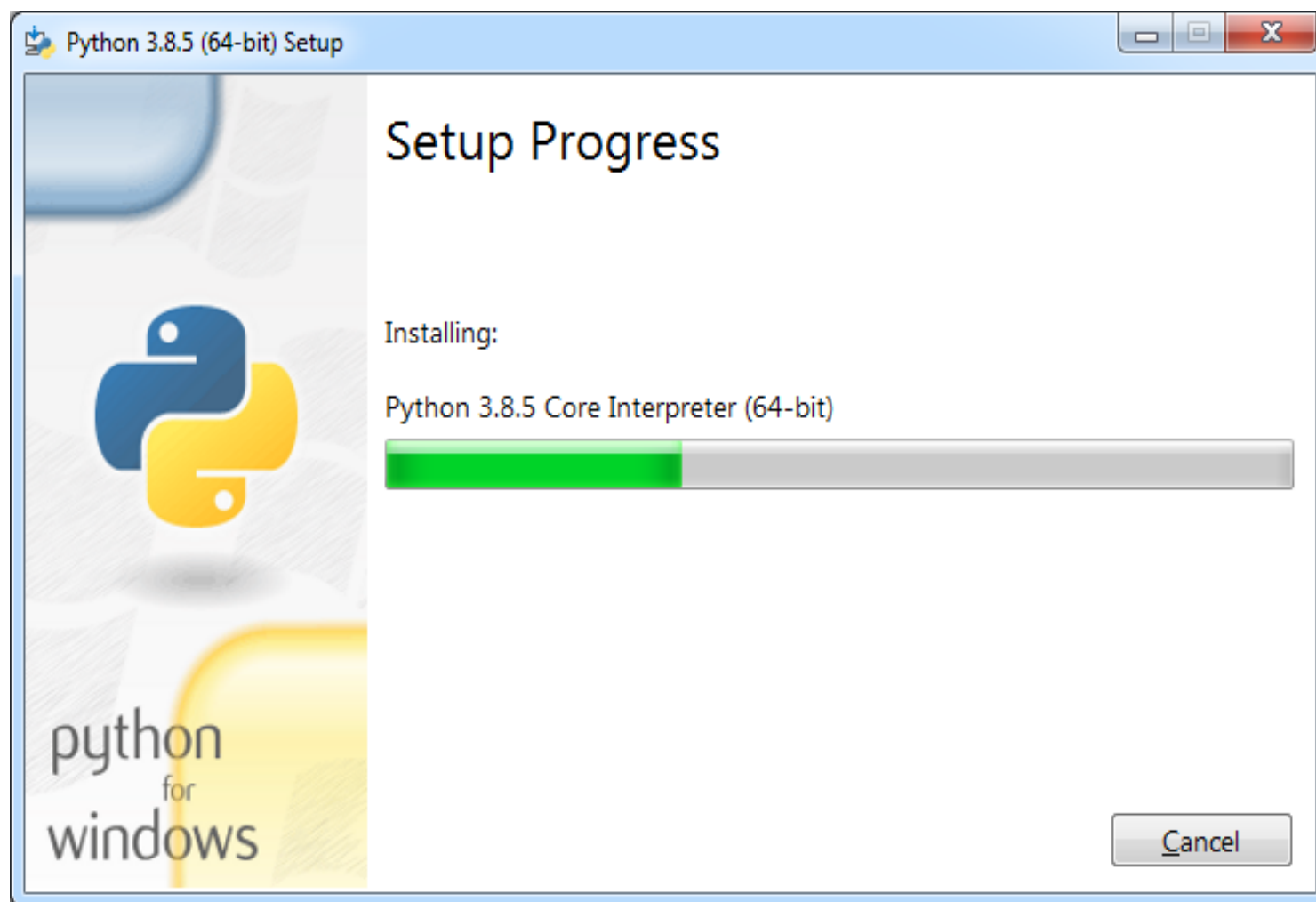
While installing, make sure you select the **Install launcher for all users** and **Add Python 3.8.5 to PATH** checkboxes. The latter places the interpreter in the execution path. For older versions of Python that do not support the **Add Python to Path** checkbox.

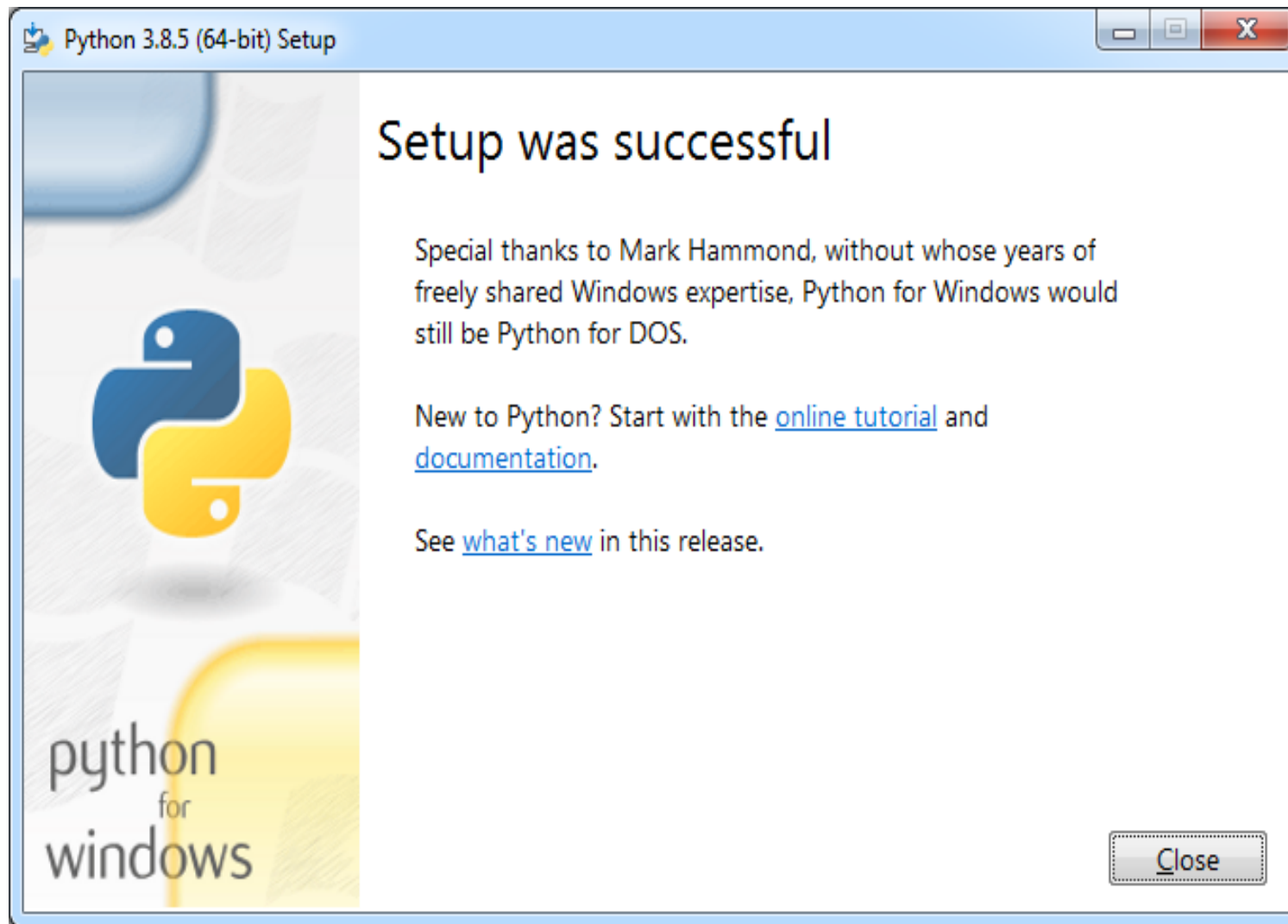




Click on Yes button

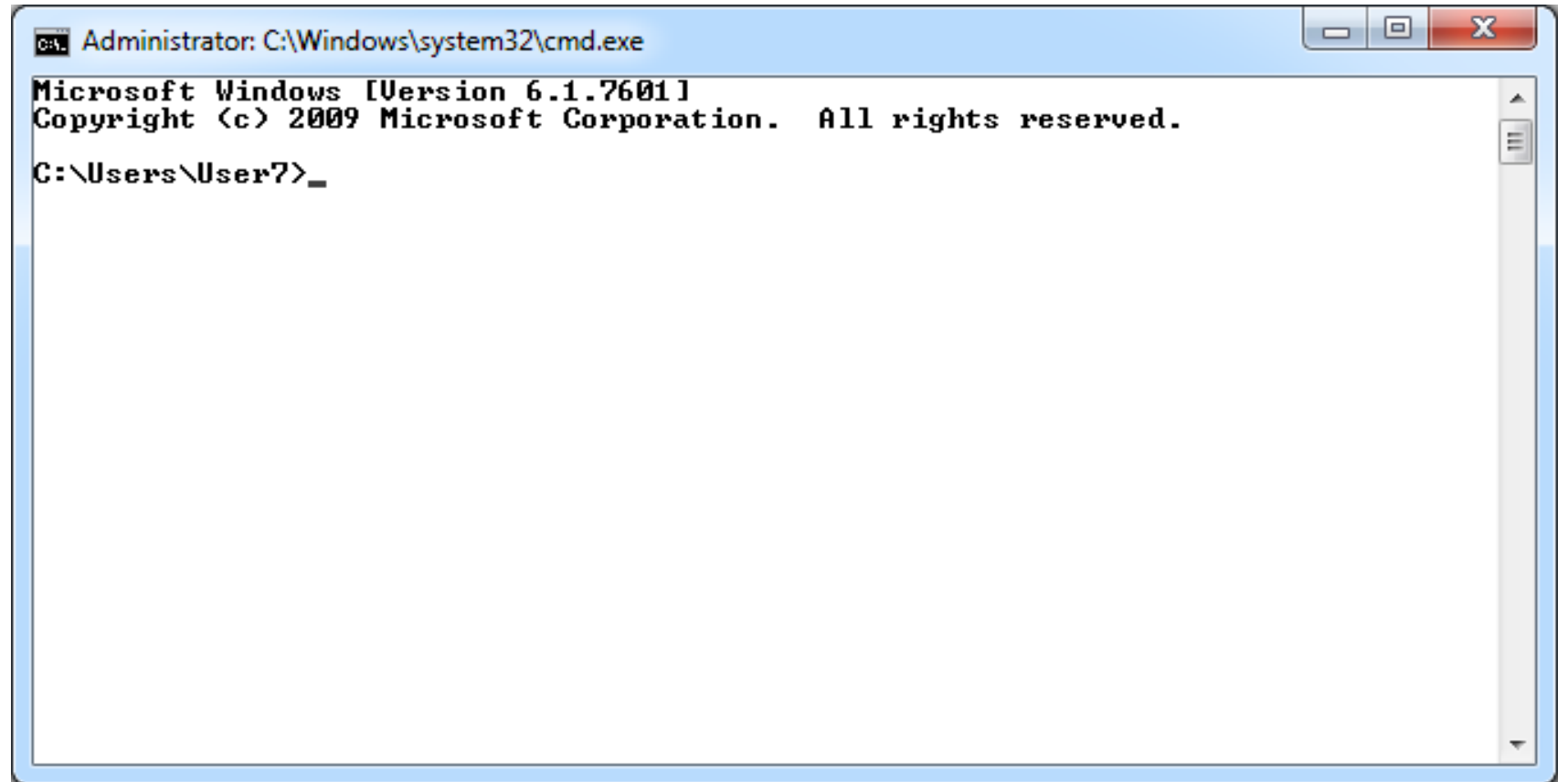
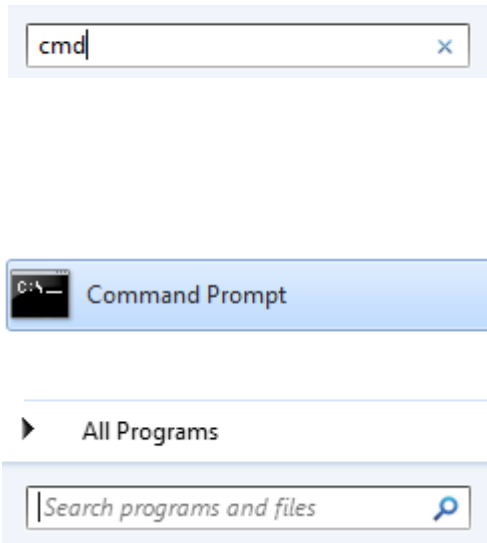




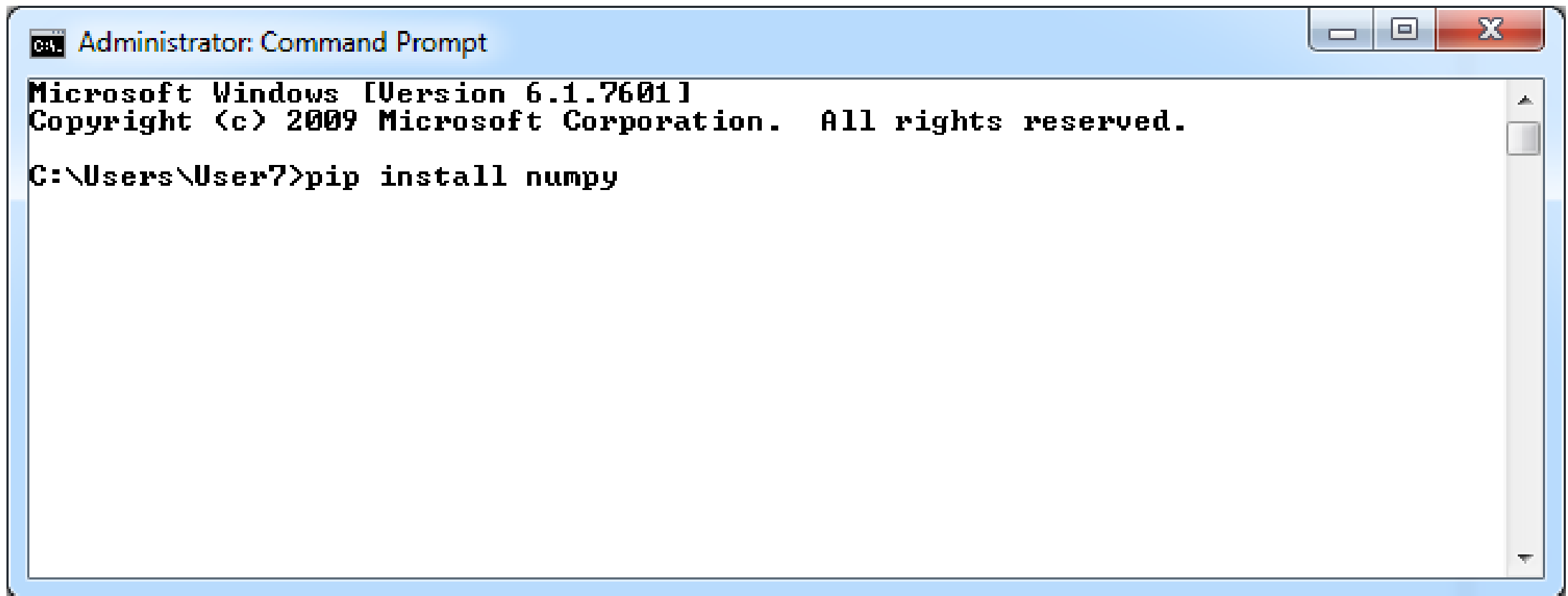


# COMMAND PROMPT

Press Windows Key + R, type cmd and Enter



# INSTALL PACKAGES



A screenshot of a Windows Command Prompt window titled "Administrator: Command Prompt". The window has a blue title bar and standard Windows window controls (minimize, maximize, close) in the top right corner. The command prompt shows the following text:

```
Microsoft Windows [Version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
  
C:\Users\User7>pip install numpy
```

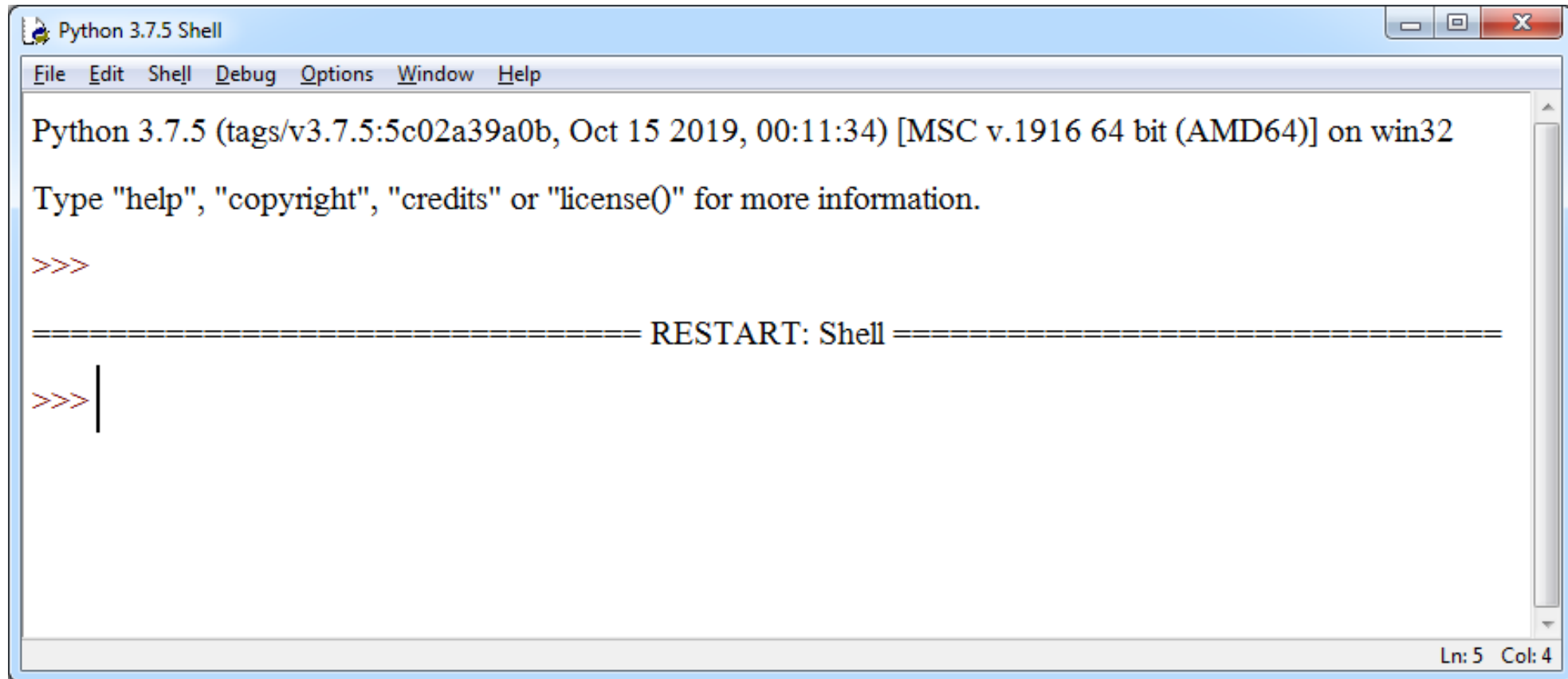
The text is displayed in a monospaced font. The prompt character is a green greater-than sign. The command 'pip install numpy' is entered on the line following the prompt. The window has a vertical scrollbar on the right side.

# INSTALLING PACKAGES

```
pip install --upgrade pip
pip install numpy
pip install matplotlib
pip install scipy
pip install joblib
pip install -U scikit-learn
pip install pygame
pip install opencv-python
pip install opencv-contrib-python
pip install Pillow
pip install pandas
pip install nltk
pip install keras
pip install pyinstaller
```



# PYTHON SHELL WINDOW (START->(IDLE PYTHON...))

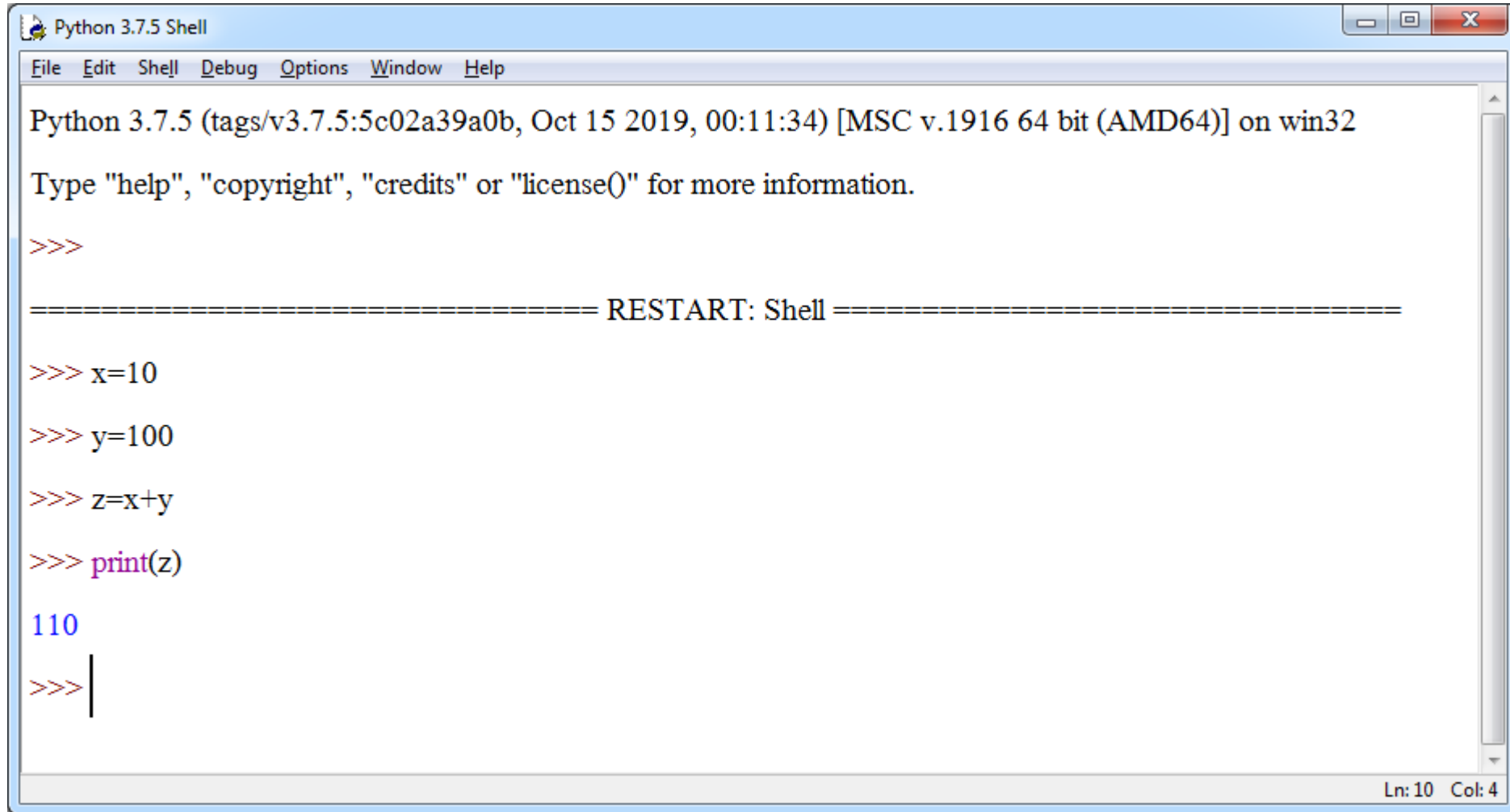
A screenshot of the Python 3.7.5 Shell window. The window has a title bar that says "Python 3.7.5 Shell" and standard Windows window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Window, and Help. The main text area contains the following text: "Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32", followed by "Type 'help', 'copyright', 'credits' or 'license()' for more information.", then a red prompt ">>>" on a new line. The next line is a separator consisting of a dashed line, the text "RESTART: Shell", and another dashed line. Below this is another red prompt ">>>" followed by a vertical cursor bar. The status bar at the bottom right shows "Ln: 5 Col: 4".

```
Python 3.7.5 Shell
File Edit Shell Debug Options Window Help
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: Shell =====
>>> |
```

Ln: 5 Col: 4



# WRITE A PROGRAM

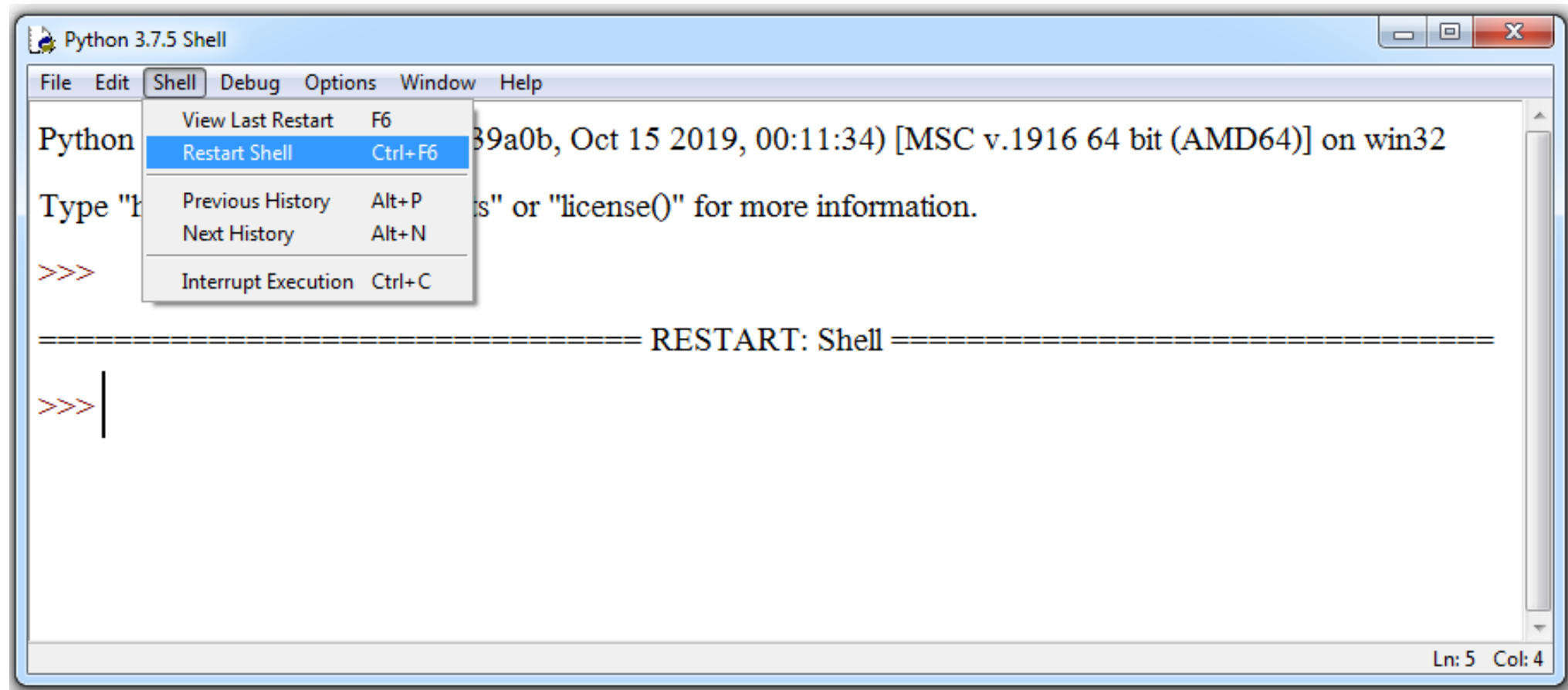


The image shows a screenshot of a Python 3.7.5 Shell window. The window has a title bar that says "Python 3.7.5 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area contains the following text:

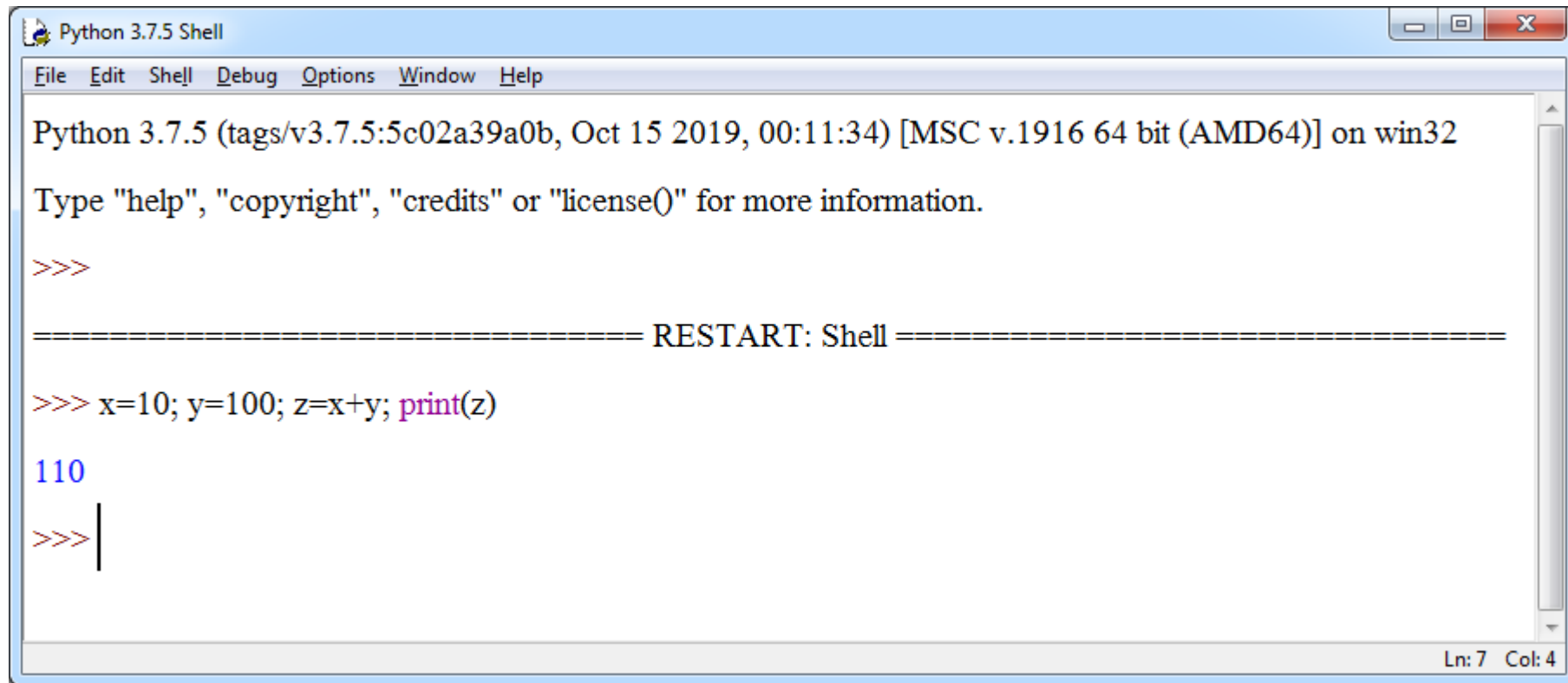
```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: Shell =====
>>> x=10
>>> y=100
>>> z=x+y
>>> print(z)
110
>>> |
```

The status bar at the bottom right of the window shows "Ln: 10 Col: 4".

# PYTHON SHELL WINDOW (SHELL > RESTART SHELL)



# PYTHON SHELL WINDOW (CONT.)



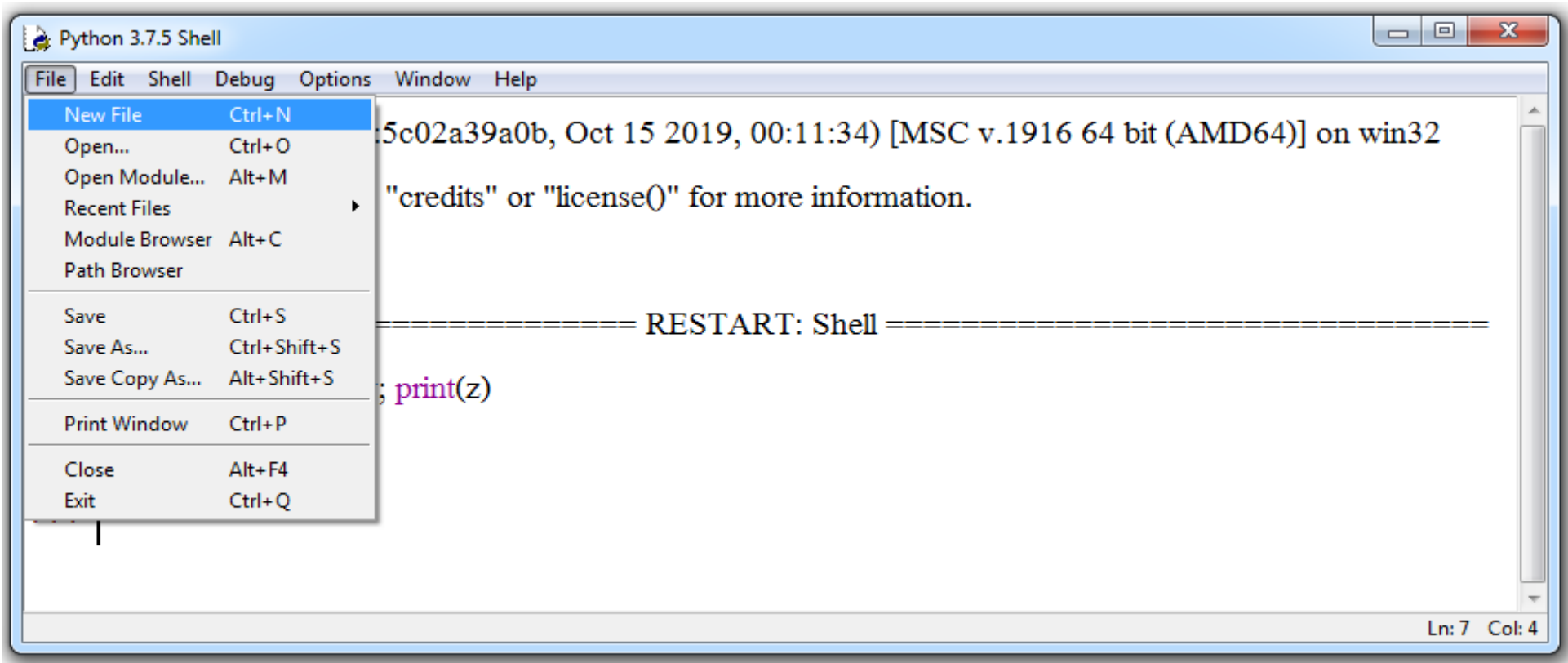
The screenshot shows a 'Python 3.7.5 Shell' window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The main text area displays the following content:

```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

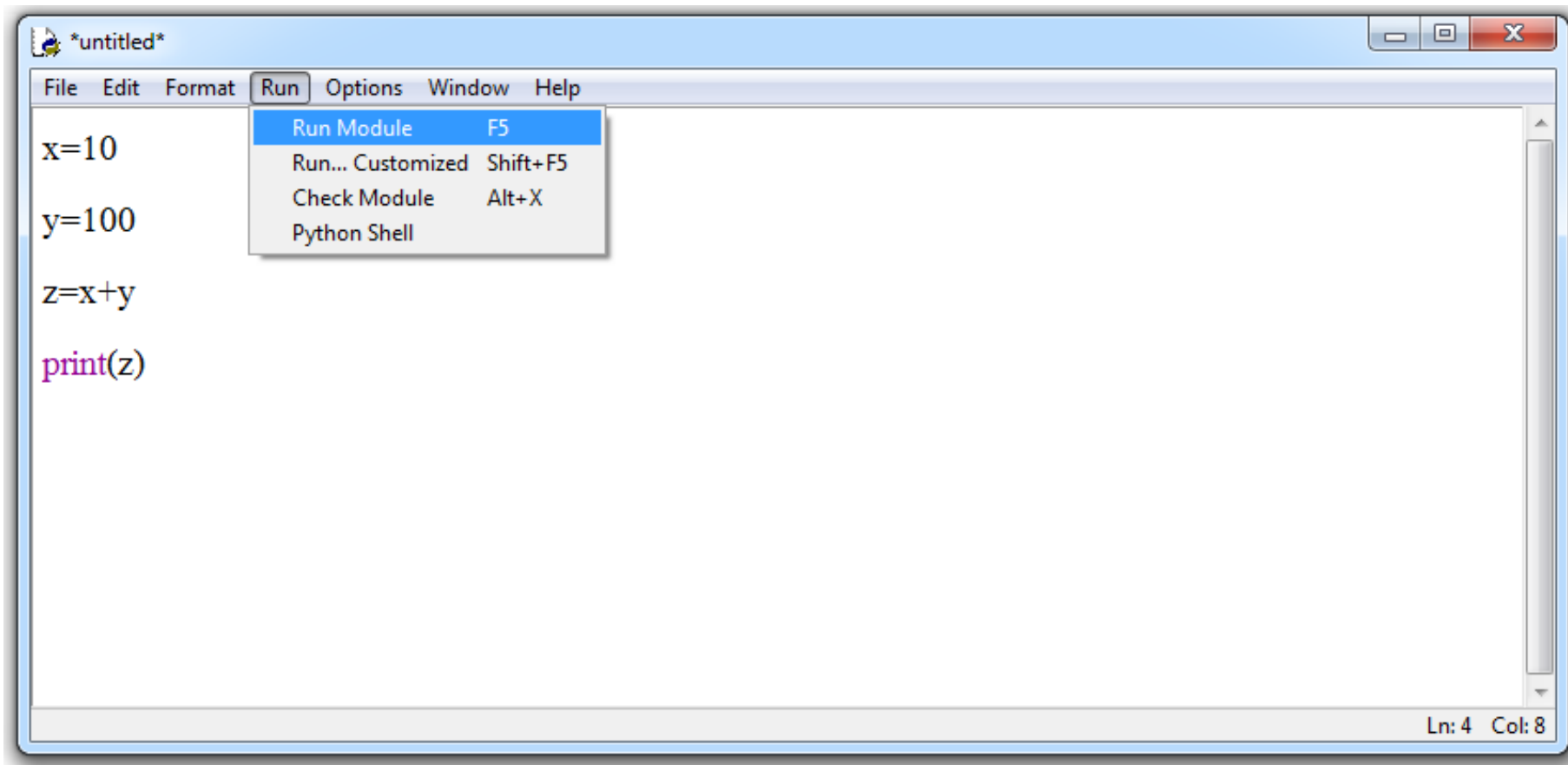
===== RESTART: Shell =====
>>> x=10; y=100; z=x+y; print(z)
110
>>> |
```

The status bar at the bottom right indicates 'Ln: 7 Col: 4'.

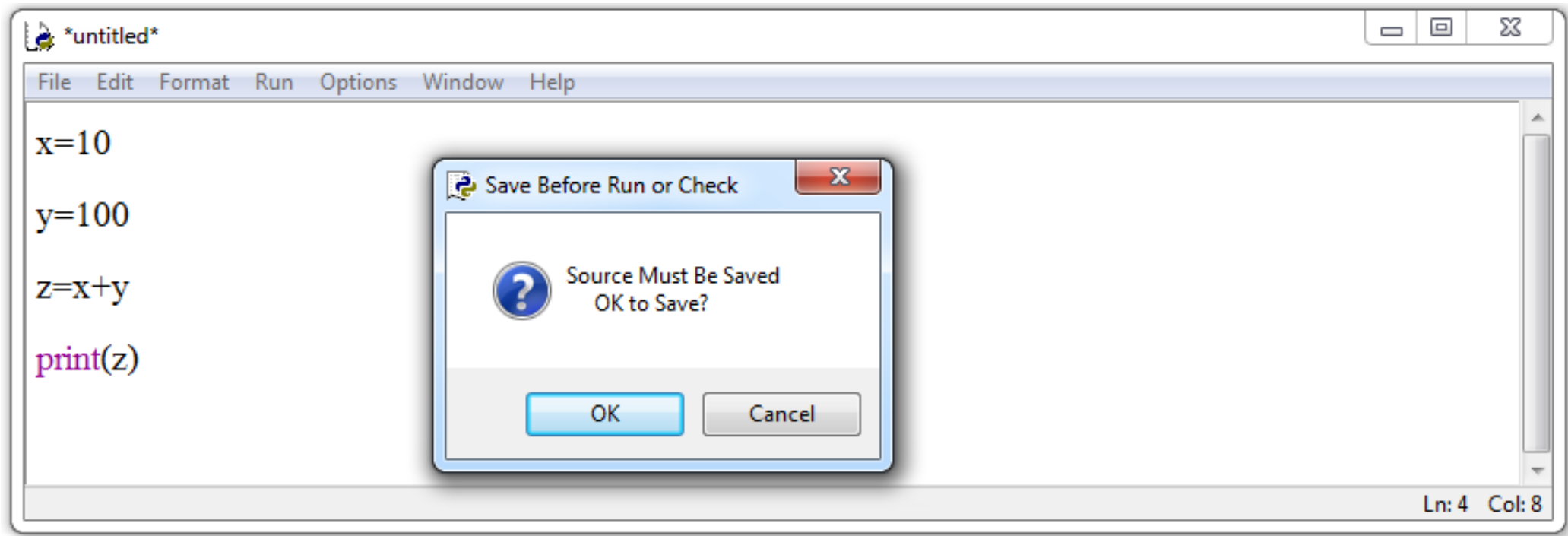
# PYTHON EDIT WINDOW



# PYTHON EDIT WINDOW (RUN MODULE F5)

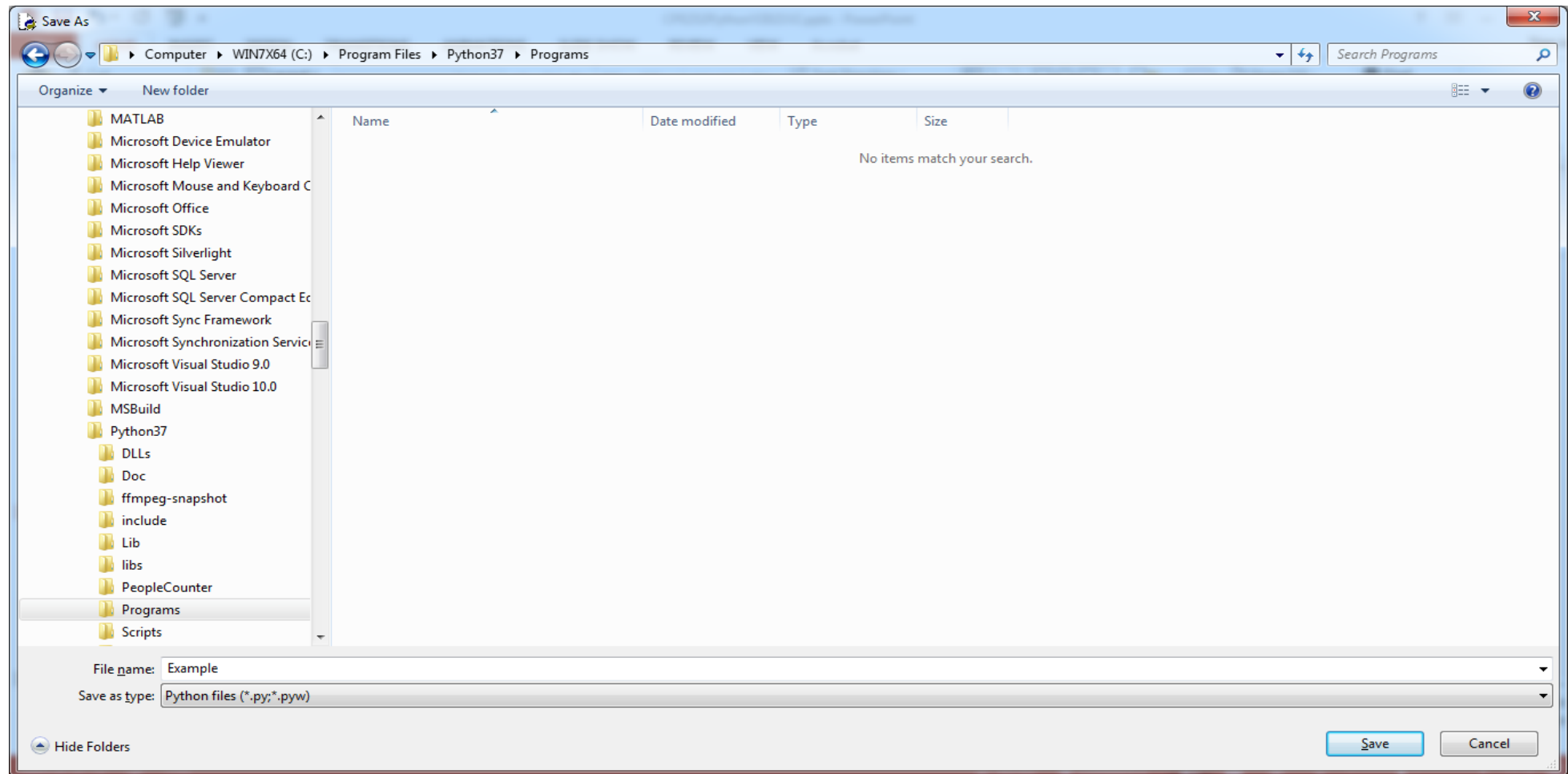


# PYTHON EDIT WINDOW (CONT.) (SAVE THE PROGRAM)

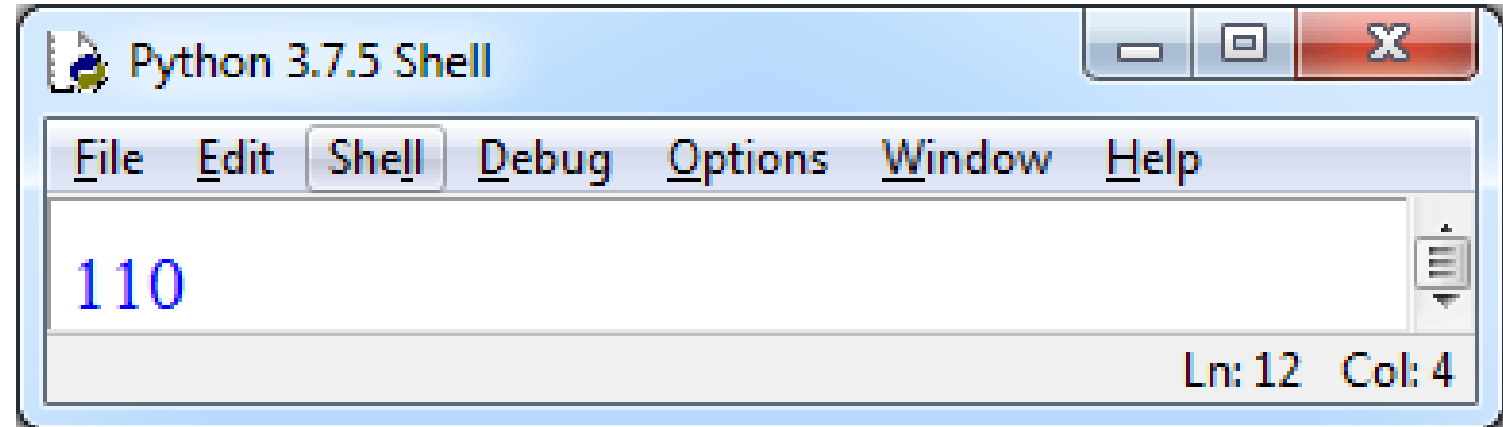
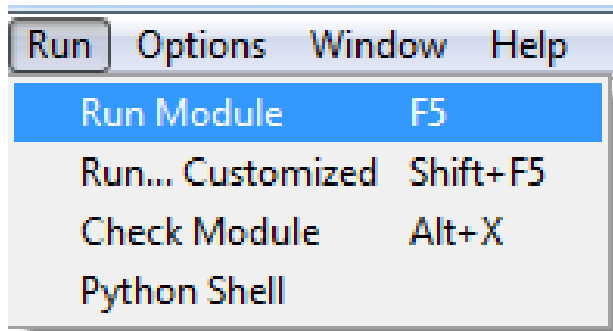




# PYTHON EDIT WINDOW (CONT.) (SAVE THE PROGRAM)



# OUTPUT



# PYTHON PROGRAM TO PRINT HELLO WORLD!

Traditionally, the first program you write in a new language is called “Hello, World!” because all it does is display the words “Hello, World!” In Python, it looks like this:

```
>>> print('Hello, World!')
```

This is an example of a **print statement**, although it doesn’t actually print anything on paper. It displays a result on the screen. In this case, the result is the words Hello, World!

# PYTHON PROGRAM TO PRINT HELLO WORLD!

```
print('Hello, World!')  
print('Hello, World!')
```

Hello, World!

Hello, World!

```
print('Hello, World!', end='')  
print('Hello, World!', end='')
```

Hello, World! Hello, World!

# PYTHON PROGRAM TO PRINT HELLO WORLD!

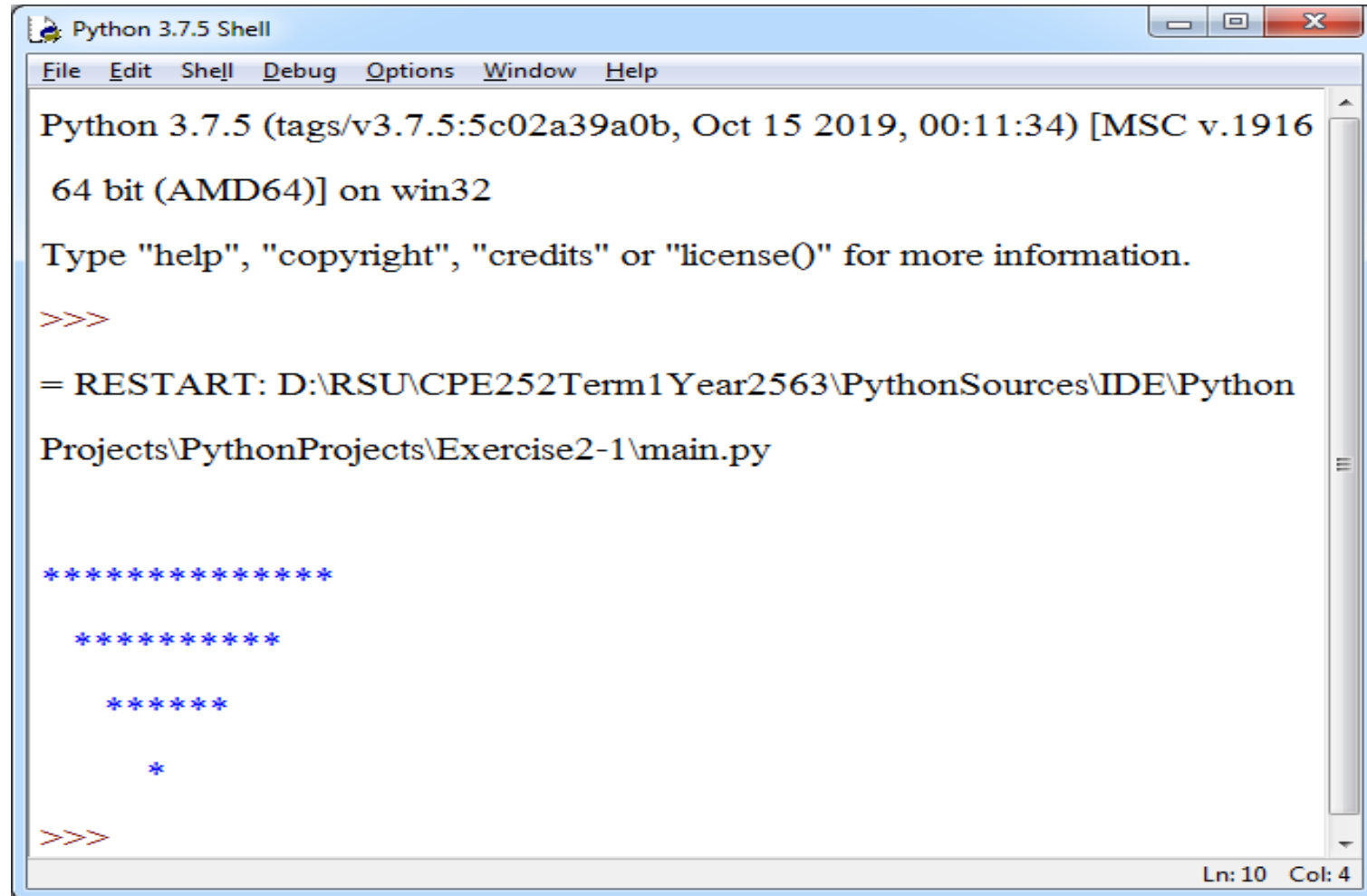
In this program, we have used the built-in `print()` function to print the string Hello, world! on our screen.

By the way, a string is a sequence of characters. In Python, strings are enclosed inside single quotes, double quotes, or triple quotes.

# PROGRAM

```
print()  
print('*****')  
print('    *****')  
print('        *****')  
print('            *')
```

# OUTPUT



A screenshot of a Python 3.7.5 Shell window. The window has a title bar that says "Python 3.7.5 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following output:

```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916  
64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
= RESTART: D:\RSU\CPE252Term1Year2563\PythonSources\IDE\Python  
Projects\PythonProjects\Exercise2-1\main.py  
  
*****  
    *****  
        *****  
            *  
  
>>>
```

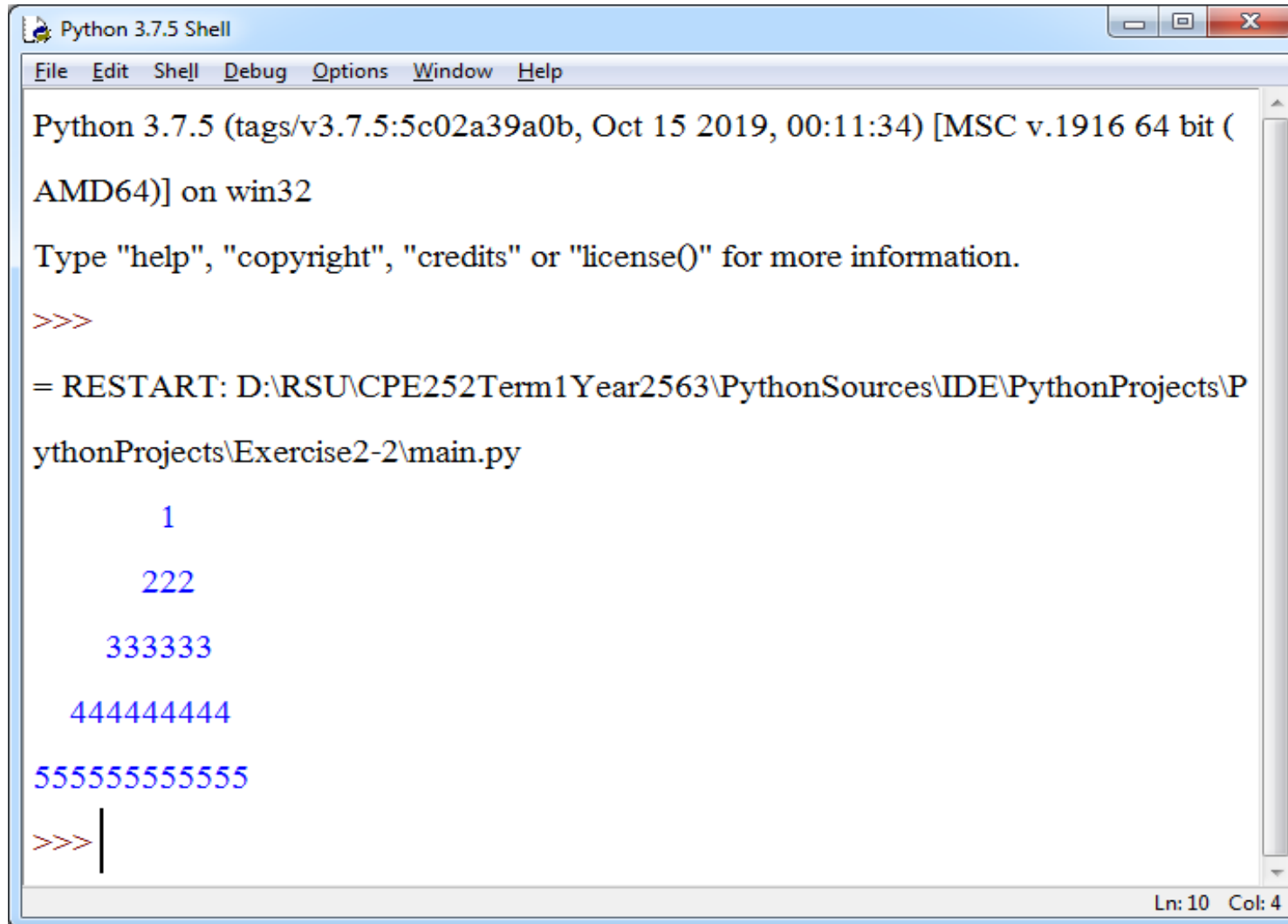
The status bar at the bottom right indicates "Ln: 10 Col: 4".



# PROGRAM

```
print (  
    '          1',  
    '        222',  
    '      333333',  
    '    444444444',  
    '5555555555555',  
    sep=' \n' )
```

# OUTPUT

A screenshot of a Python 3.7.5 Shell window. The window has a title bar that says "Python 3.7.5 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following output:

```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
= RESTART: D:\RSU\CPE252Term1Year2563\PythonSources\IDE\PythonProjects\PythonProjects\Exercise2-2\main.py  
  
1  
222  
333333  
4444444444  
555555555555  
>>> |
```

The output consists of the Python version and environment information, a prompt for help, a restart message with the file path, and five lines of output: "1", "222", "333333", "4444444444", and "555555555555". The prompt ">>>" is followed by a vertical bar "|". The status bar at the bottom right shows "Ln: 10 Col: 4".

# PROGRAM

```
print(  
    '        1',  
    '        222',  
    '        333333',  
    '    4444444444',  
    ' 55555555555555',  
    sep="")
```

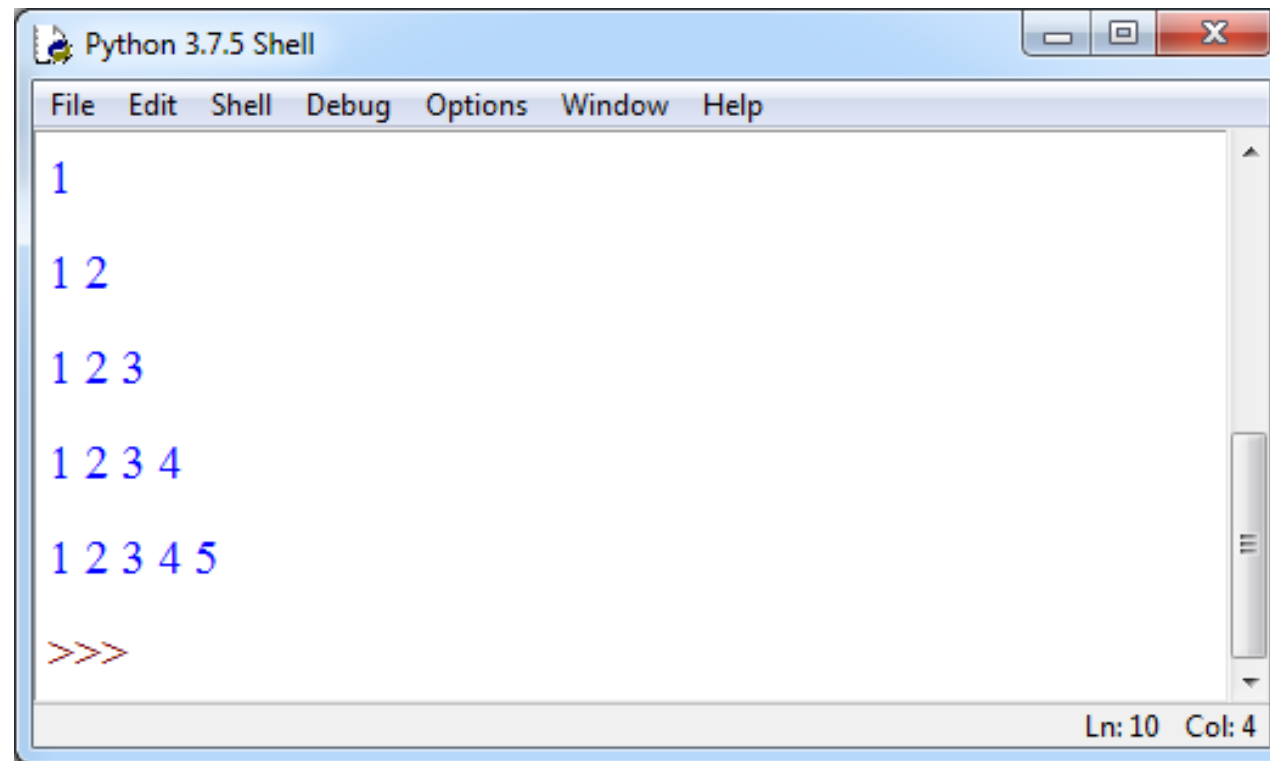
Output

1          222          333333    4444444444    55555555555555

# PROGRAM

```
print('1')  
print('1 2')  
print('1 2 3')  
print('1 2 3 4')  
print('1 2 3 4 5')
```

# OUTPUT



A screenshot of a Python 3.7.5 Shell window. The window has a title bar with the text "Python 3.7.5 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Window, and Help. The main area of the window is a text editor showing the output of a program. The output consists of five lines of blue text: "1", "1 2", "1 2 3", "1 2 3 4", and "1 2 3 4 5". Below these lines is a red prompt character ">>>". The status bar at the bottom right of the window shows "Ln: 10 Col: 4".

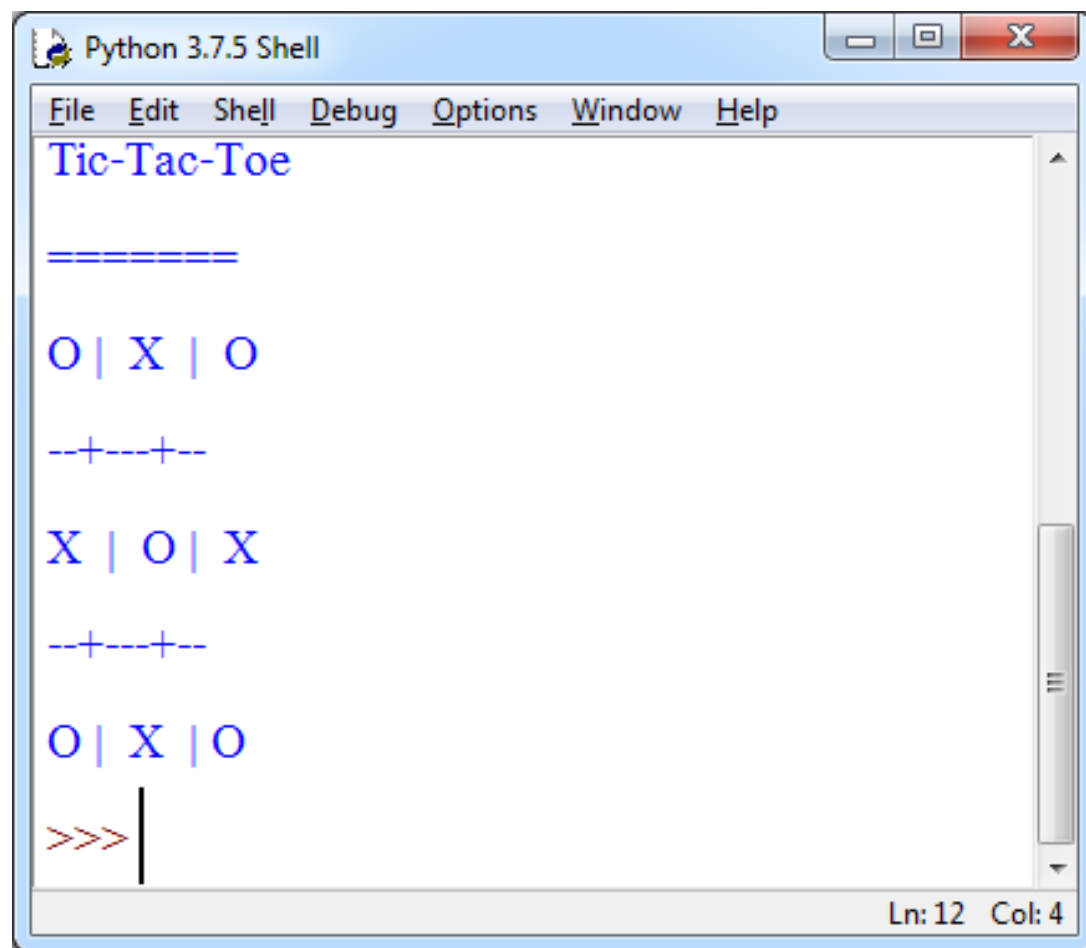
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
>>>
```

Ln: 10 Col: 4

# PROGRAM

```
print('Tic-Tac-Toe')  
print('=====')  
  
print('O | X | O')  
print('--+---+--')  
print('X | O | X ')  
print('--+---+--')  
print('O | X | O')
```

# OUTPUT



A screenshot of a Python 3.7.5 Shell window. The window has a title bar with the text "Python 3.7.5 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Shell, Debug, Options, Window, and Help. The main text area displays the output of a Tic-Tac-Toe program. The text is as follows:

```
Tic-Tac-Toe  
  
=====
```

O	X	O
---	---	---

```
--+---+--  
  


|   |   |   |
|---|---|---|
| X | O | X |
|---|---|---|

  
--+---+--  
  

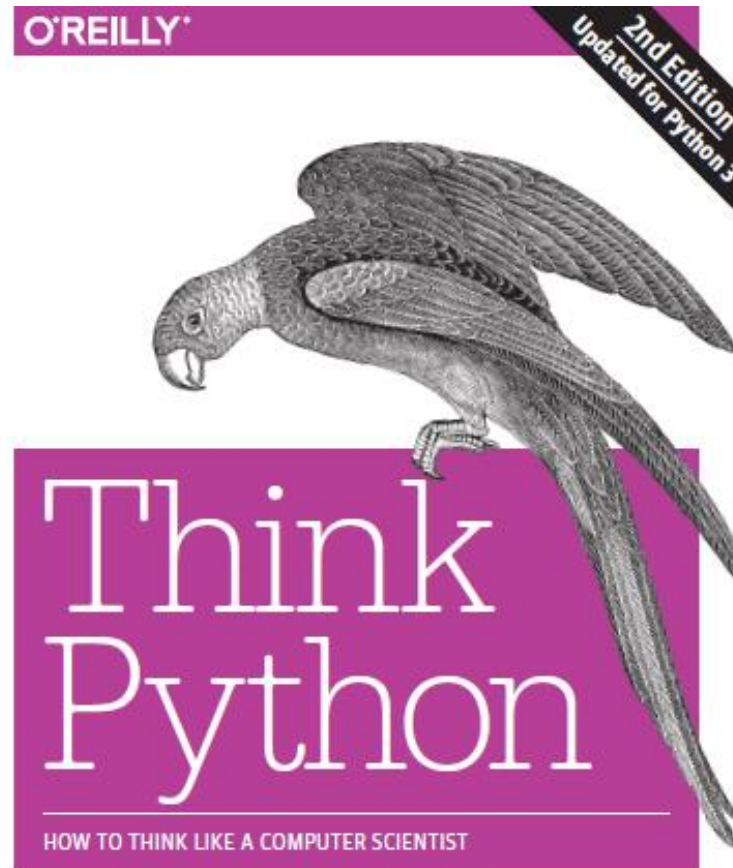

|   |   |   |
|---|---|---|
| O | X | O |
|---|---|---|

  
>>> |
```

The status bar at the bottom right of the window shows "Ln: 12 Col: 4".



# BOOKS



Allen B. Downey

# VALUES AND TYPES

A **value** is one of the basic things a program works with, like a letter or a number.

Some values we have seen so far are 2, 42.0, and 'Hello, World!'  
These values belong to different **types**:

2 is an **integer**,  
42.0 is a **floating-point number**, and  
'Hello, World!' is a **string**,

so-called because the letters it contains are strung together.

# WHAT TYPE A VALUE HAS

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
```

```
<class 'int'>
```

```
>>> type(42.0)
```

```
<class 'float'>
```

```
>>> type('Hello, World!')
```

```
<class 'str'>
```

# ASSIGNMENT STATEMENTS

An **assignment statement** creates a new variable and gives it a value:

```
>>> message = 'And now for something completely different'
```

```
>>> n = 17
```

```
>>> pi = 3.141592653589793
```

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  $\pi$  to pi.

# VARIABLE NAMES

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total\_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- The interpreter uses **keywords** to recognize the structure of the program, and they cannot be used as variable names

# VARIABLE NAMES

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be as long as you like. They can contain both letters and numbers, but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lowercase for variables names.

The underscore character, `_`, can appear in a name. It is often used in names with multiple words, such as `your_name` or `airspeed_of_unladen_swallow`.

# EXAMPLES OF ILLEGAL VARIABLE NAMES

```
>>> 76trombones = 'big parade'
```

```
SyntaxError: invalid syntax
```

```
>>> more@ = 1000000
```

```
SyntaxError: invalid syntax
```

```
>>> class = 'Advanced Theoretical Zymurgy'
```

```
SyntaxError: invalid syntax
```





# EXAMPLES OF ILLEGAL VARIABLE NAMES

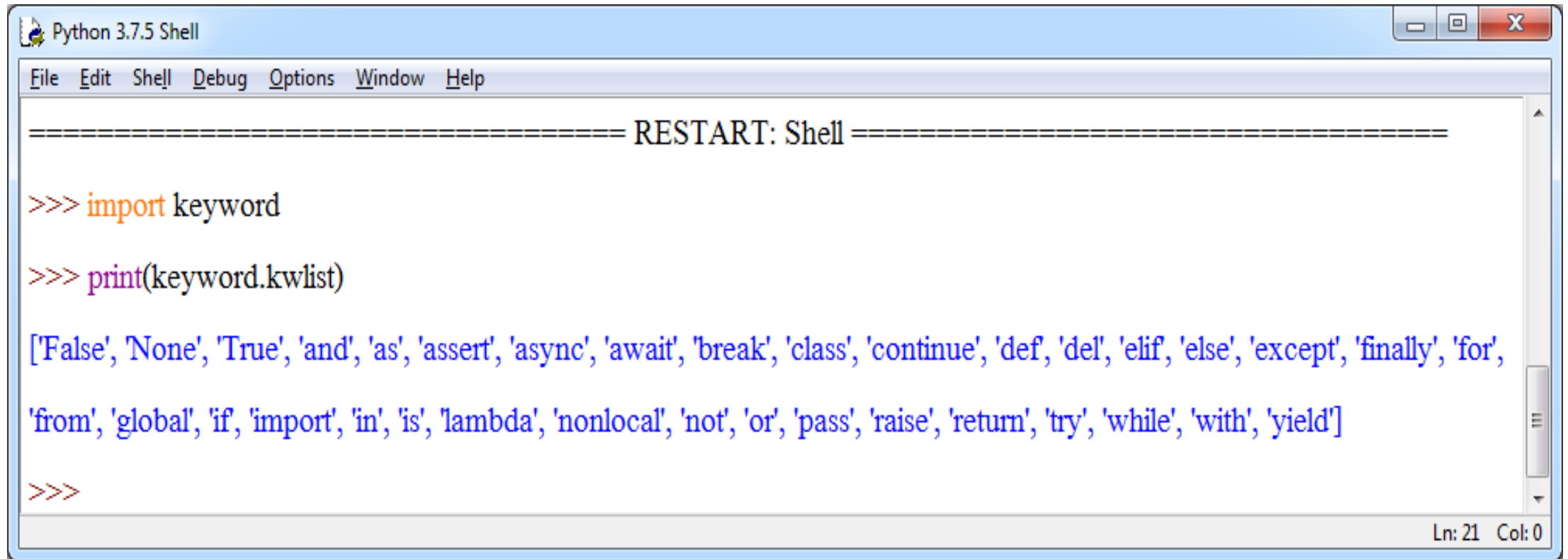
76trombones is illegal because it begins with a number.

more@ is illegal because it contains an illegal character, @.

# PYTHON 3 KEYWORDS

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

# PYTHON KEYWORDS (RESERVED WORDS)



```
Python 3.7.5 Shell
File Edit Shell Debug Options Window Help
===== RESTART: Shell =====
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Ln: 21 Col: 0

# OPERATOR

```
>>> 40 + 2
```

```
42
```

```
>>> 43 - 1
```

```
42
```

```
>>> 6 * 7
```

```
42
```

The operator / performs (float) division:

```
>>> 84 / 2
```

```
42.0
```

# OPERATOR

the operator `**` performs exponentiation; that is, it raises a number to a power:

```
>>> 6**2 + 6
```

```
42
```

# EXPRESSIONS AND STATEMENTS

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:

```
>>> 42
```

```
42
```

```
>>> n
```

```
17
```

```
>>> n + 25
```

```
42
```

# EXPRESSIONS AND STATEMENTS

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17  
>>> print(n)
```

The first line is an assignment statement that gives a value to `n`.  
The second line is a print statement that displays the value of `n`.



# SCRIPT

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.

# USING VARIABLES

```
x = 123  
y = 456.789
```

```
a = -10  
b=-20.25
```

A python variable is not strict to any data type.

```
x=123  
x=4.56
```

```
y=7.11  
y=101
```

# FORMAT STRING (1)

```
firstname = 'Rong'  
print('My name is %s.' % firstname)
```

My name is Rong.

# FORMAT STRING (1)

```
firstname = 'Rong'  
num = 99.99  
print('My name is %s %f.' % (firstname, num))
```

My name is Rong 99.990000.

# FORMAT STRING (1)

```
firstname = 'Rong'  
beverage = 'tea'  
drink_time = 1  
money = 50  
print('My name is %s. I like a %s and drink it %d time(s) a  
day.' % (firstname, beverage, drink_time))
```

My name is Rong. I like a tea and drink it 1 time(s) a day.

# FORMAT STRING (2)

```
firstname = 'Rong'  
beverage = 'tea'  
drink_time = 1  
money = 50
```

```
result_message = 'My name is {}. I like a {} and drink it {}  
time(s) a day.'.format(firstname, beverage, drink_time)  
print(result_message)
```

My name is Rong. I like a tea and drink it 1 time(s) a day.

# FORMAT STRING (2)

```
firstname = 'Rong'  
beverage = 'tea'  
drink_time = 1  
money = 50  
result_message = 'My name is {}. I like a {} and drink it {}  
time(s) a day.'.format(firstname, beverage, drink_time)  
print(result_message)
```

My name is Rong. I like a tea and drink it 1 time(s) a day.



# FORMAT STRING (2)

```
firstname = 'Rong'
beverage = 'tea'
drink_time = 1
money = 50
spend_money = money * drink_time

result_message = 'My name is {0}. I like a {1} and drink it {2}
time(s) a day. The {1} is {3} Baht. I\'m spend {4} Baht per a
day.'.format(firstname, beverage, drink_time, money, spend_money)

print(result_message)
```

My name is Rong. I like a tea and drink it 1 time(s) a day. The tea is 50 Baht. I'm spend 50 Baht per a day.

# FORMAT STRING (3)

```
firstname = 'Rong'  
beverage = 'tea'  
drink_time = 1  
money = 50
```

```
print(f'My name is {firstname}. I like a {beverage} and drink  
it {drink_time} time(s) a day. The {beverage} is {money} Baht.  
I\'m spend {money * 2} Baht per a day.')
```

My name is Rong. I like a tea and drink it 1 time(s) a day. The tea is 50 Baht. I'm spend 100 Baht per a day.

# FORMAT STRING

Old

```
'%s %s' % ('one', 'two')
```

New

```
'{} {}'.format('one', 'two')
```

Output

```
o n e   t w o
```

Old

```
'%d %d' % (1, 2)
```

New

```
'{} {}'.format(1, 2)
```

Output

```
1 2
```

# FORMAT STRING

New

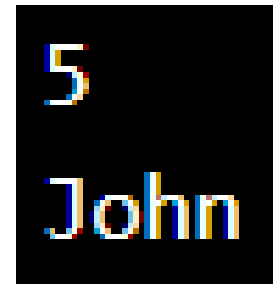
```
'{1} {0}'.format('one', 'two')
```

Output

```
t w o   o n e
```

# VARIABLES (A VALUE TO A VARIABLE)

```
x = 5  
y = "John"  
print(x)  
print(y)
```



5  
John

# VARIABLES (A VALUE TO A VARIABLE)

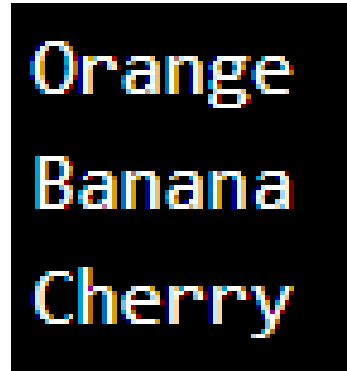
```
x = 4          # x is of type int  
x = "Sally"    # x is now of type str  
print(x)
```



Sally

# MANY VALUES TO MULTIPLE VARIABLES

```
x, y, z = "Orange", "Banana", "Cherry"  
print(x)  
print(y)  
print(z)
```



Orange  
Banana  
Cherry



# ONE VALUE TO MULTIPLE VARIABLES

```
x = y = z = "Orange"  
print(x)  
print(y)  
print(z)
```

# COMPUTE THE TOTAL NUMBER OF SECONDS

```
h = int(input())           10
m = int(input())           20
s = int(input())           40
total = 60*60*h + 60*m + s
print(total)               37240
```

10 hour(s), 20 minute(s) and 40 second(s) equal 37240 second(s)

# COMPUTE THE TOTAL NUMBER OF SECONDS

```
h = int(input('Enter hours: '))
m = int(input('Enter minutes: '))
s = int(input('Enter seconds: '))
total = 60*60*h + 60*m + s
print(f'{h} hour(s), {m} minute(s) and {s} second(s) equal {total} second(s).')
```

Enter hours: 10

Enter minutes: 5

Enter seconds: 2

10 hour(s), 5 minute(s) and 2 second(s) equal 36302 second(s).

# CIRCUMFERENCE

```
radius=float(input('Radius: '))  
circumference=2*3.14159*radius  
print(f'Radius of the circle is {radius} and circumference of the circle  
is {circumference:.2f} m.')
```

# AVERAGE OF 4 NUMBERS

```
sum = 0
```

```
sum += int(input('Number#1: '))
```

```
sum += int(input('Number#2: '))
```

```
sum += int(input('Number#3: '))
```

```
sum += int(input('Number#4: '))
```

```
average = sum / 4
```

```
print('\nSum:', sum)
```

```
print('Average:', average)
```

```
sum += int(input('Number#1: '))  
is a shorthand notation of  
sum = sum + int(input('Number#1: '))
```

Sum of the numbers is ..... and the average is .....

# AVERAGE OF 4 NUMBERS

```
sum = 0
```

```
sum += float(input('Number#1: '))
```

```
sum += float(input('Number#2: '))
```

```
sum += float(input('Number#3: '))
```

```
sum += float(input('Number#4: '))
```

```
average = sum / 4
```

```
print('Sum of the numbers is %f and the average is %f.' %(sum, average))
```

Number#1: 10

Number#2: 20

Number#3: 30

Number#4: 40

Sum of the numbers is 100.000000 and the average is 25.000000.

# ORDER OF OPERATIONS

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.



# ORDER OF OPERATIONS

**Exponentiation** has the next highest precedence, so  $1 + 2^{**}3$  is 9, not 27, and  $2^{*}3^{**}2$  is 18, not 36.

**Multiplication** and **Division** have higher precedence than **Addition** and **Subtraction**. So  $2^{*}3-1$  is 5, not 4, and  $6+4/2$  is 8, not 5.

# ORDER OF OPERATIONS

Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression  $\text{degrees} / 2 * \pi$ , the division happens first and the result is multiplied by  $\pi$ . To divide by  $2\pi$ , you can use parentheses or write  $\text{degrees} / 2 / \pi$ .

If I can't tell by looking at the expression, I use parentheses to make it obvious.

# PYTHON ASSIGNMENT OPERATORS

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

# % MODULUS

Divides left hand operand by right hand operand and returns a remainder

$$17\%3 = 2$$

# OPERATOR //

Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)

$9//2 = 4$  and  $9.0//2.0 = 4.0$ ,

$-11//3 = -4$ ,  $-11.0//3 = -4.0$

## **\*\* EXPONENT**

Performs exponential (power) calculation on operators

$$3^{**}5=243$$

# PYTHON PROGRAM TO ADD TWO NUMBERS

```
# This program adds two numbers
```

```
num1 = 1.5
```

```
num2 = 6.3
```

```
# Add two numbers
```

```
sum = num1 + num2
```

```
# Display the sum
```

```
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

The sum of 1.5 and 6.3 is 7.8



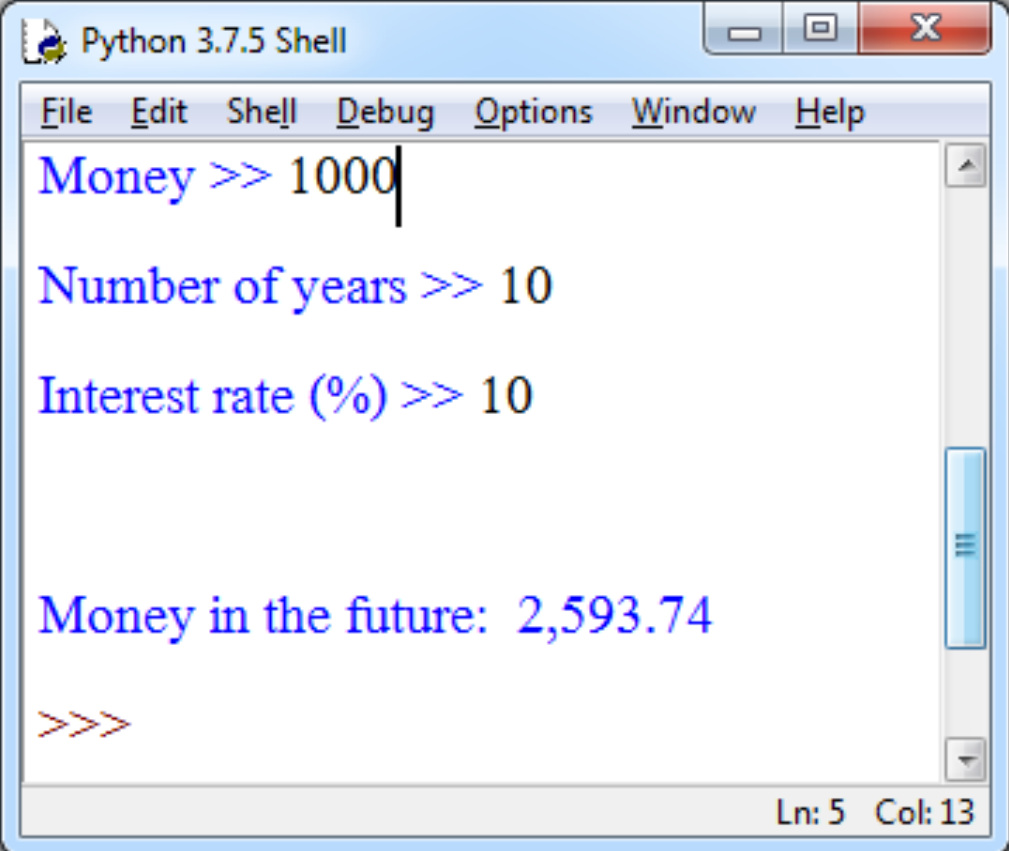
# MONEY AND INTEREST

เงิน และ ดอกเบี้ย

Money in the future = Money  $\times (1 + \text{Interest rate}/100)^{\text{year}}$

```
amount = float(input('Money >> '))
years = int(input('Number of years >> '))
rate = float(input('Interest rate (%) >> '))
fv = amount * (1 + rate/100) ** years

print('\nMoney in the future: %.2f.' %fv)
```



The screenshot shows a Python 3.7.5 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The input and output of the program are as follows:

```
Money >> 1000
Number of years >> 10
Interest rate (%) >> 10

Money in the future: 2,593.74

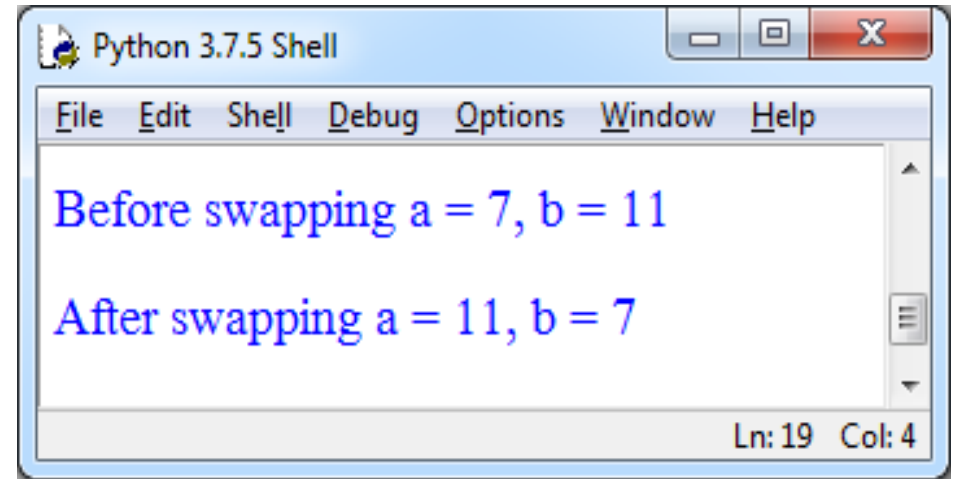
>>>
```

The status bar at the bottom right indicates "Ln: 5 Col: 13".

# SWAPPING 2 VALUES

```
a = 7
b = 11
print(f'Before swapping a = {a}, b = {b} ')

c = a
a = b
b = c
print(f'After swapping a = {a}, b = {b}')
```



The screenshot shows a window titled "Python 3.7.5 Shell" with a menu bar containing "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the output of the program in blue text: "Before swapping a = 7, b = 11" and "After swapping a = 11, b = 7". The status bar at the bottom right indicates "Ln: 19 Col: 4".

# BUILT-IN MATH FUNCTIONS

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

# BUILT-IN MATH FUNCTIONS

```
x = abs(-7.25)
print(x)
y = pow(4, 3)
print(y)
```

# COMPUTE A MATH EXPRESSION

$$y = 2 - x + \frac{3}{7}x^2 - \frac{5}{11}x^3 + \log_{10}(x)$$

```
import math
```

```
x = float(input('x >> '))
```

```
y = 2 - x + 3/7*x**2 - 5/11*x**3 + math.log(x,10)
```

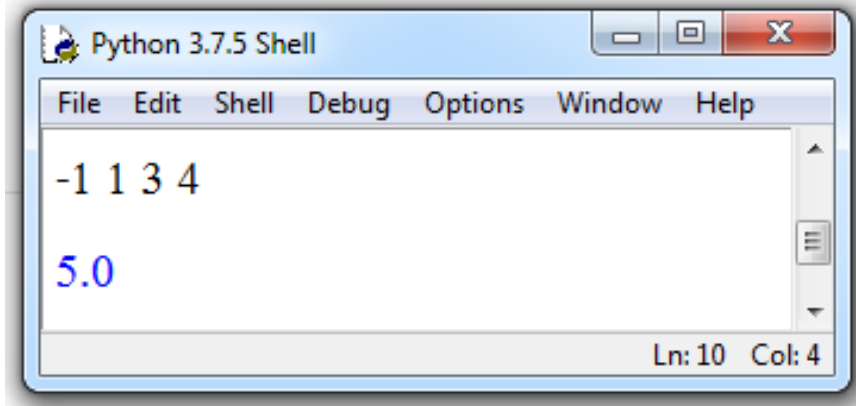
```
print(y)
```

# DISTANCE BETWEEN TWO POINTS

The distance formula is an algebraic expression used to determine the distance between two points with the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ .

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
import math
x1,y1,x2,y2 = [float(e) for e in input().split()]
d = math.sqrt((x2-x1)**2+(y2-y1)**2)
print(d)
```

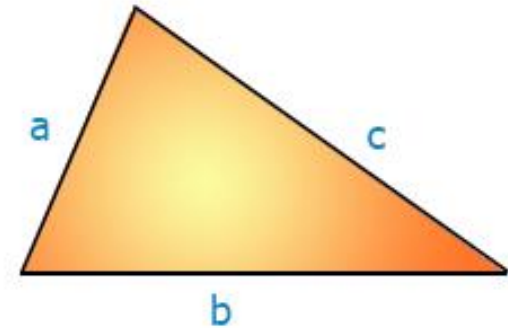


A screenshot of a Python 3.7.5 Shell window. The window has a title bar with the text 'Python 3.7.5 Shell' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area contains the input '-1 1 3 4' on the first line and the output '5.0' on the second line. The status bar at the bottom right shows 'Ln: 10 Col: 4'.

# AREA OF A SCALENE TRIANGLE

```
import math
a=float(input('Side a: '))
b=float(input('Side b: '))
c=float(input('Side c: '))
s=(a+b+c)/2
scalene_area=math.sqrt(s*(s-a)*(s-b)*(s-c))
print(f'The area of a scalene is{scalene_area}.')
```

Area of Scalene Triangle



Area of Scalene Triangle with sides  $a$ ,  $b$ , and  $c$

$$\text{Semi perimeter } s = \frac{a + b + c}{2}$$

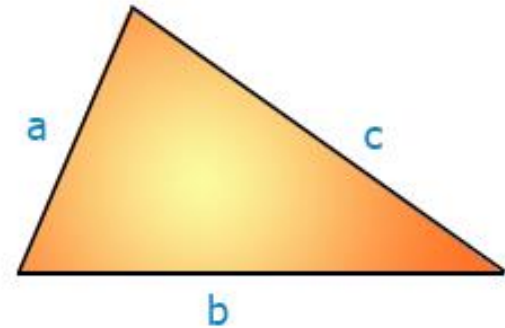
$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$



# AREA OF A SCALENE TRIANGLE

```
a=float(input('Side a: '))  
b=float(input('Side b: '))  
c=float(input('Side c: '))  
s=(a+b+c)/2  
scalene_area=(s*(s-a)*(s-b)*(s-c))**(0.5)  
print(f'The area of a scalene is{scalene_area}.')
```

Area of Scalene Triangle



Area of Scalene Triangle with sides  $a$ ,  $b$ , and  $c$

$$\text{Semi perimeter } s = \frac{a + b + c}{2}$$

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$



$$ax^2+bx+c=0$$

```
import math
```

```
a=float(input('a: '))
```

```
b=float(input('b: '))
```

```
c=float(input('c: '))
```

```
root1 = (-b+math.sqrt(b*b-4*a*c))/2*a
```

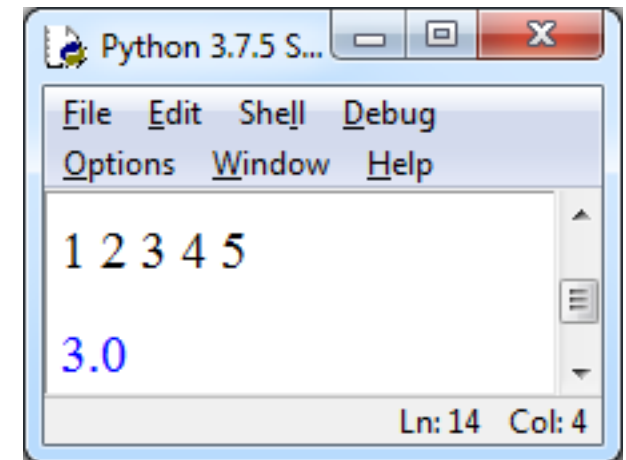
```
root2 = (-b-math.sqrt(b*b-4*a*c))/2*a
```

```
print(root1, root2)
```

# AVERAGE OF 5 NUMBERS

ค่าเฉลี่ยตัวเลข 5 ตัว

```
a,b,c,d,g = [float(e) for e in input().split()]  
avg = (a+b+c+d+g)/5  
print(avg)
```



# STRING OPERATIONS

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

```
'2'-'1' 'eggs'/'easy' 'third'*'a charm'
```

But there are two exceptions, + and \*.

# STRING OPERATIONS

The + operator performs **string concatenation**, which means it joins the strings by linking them end-to-end. For example:

```
>>> first = 'throat'  
>>> second = 'warbler'  
>>> first + second  
throatwarbler
```

# STRING OPERATIONS

The `*` operator also works on strings; it performs repetition. For example, `'Spam'*3` is `'SpamSpamSpam'`.

If one of the values 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'`.



# 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

# COMMENTS

```
# 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 execution of the program.

# COMMENTS

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 trade-off.



# LINE COMMENT (USING HASH (#))

```
# Written be Mr. CPE252  
# 9 September 2021  
# Computer Engineering
```



# ERRORS

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

# SYNTAX ERROR

“Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so  $(1 + 2)$  is legal, but  $8)$  is a **syntax error**.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.



# RUNTIME ERROR

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.



# SEMANTIC ERROR

The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.



# FUNCTIONS

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name.

# FUNCTION CALLS

We have already seen one example of a **function call**:

```
>>> type(42)  
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

42 is an argument.

# FUNCTION CALLS

It is common to say that a function “takes” an argument and “returns” a result. The result is also called the **return value**.

Python provides functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
```

```
32
```

```
>>> int(32.5)
```

```
32
```

```
>>> int('Hello')
```

```
ValueError: invalid literal for int(): Hello
```



# FUNCTION CALLS

int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
```

```
3
```

```
>>> int(-2.3)
```

```
-2
```

# FUNCTION CALLS

float converts integers and strings to floating-point numbers:

```
>>> float(32)
```

```
32.0
```

```
>>> float('3.14159')
```

```
3.14159
```

Finally, str converts its argument to a string:

```
>>> str(32)
```

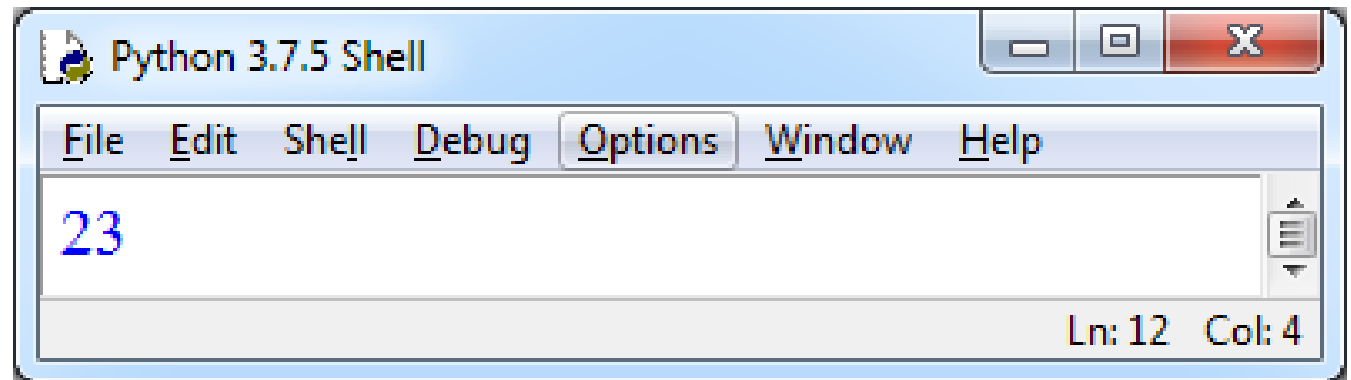
```
'32'
```

```
>>> str(3.14159)
```

```
'3.14159'
```

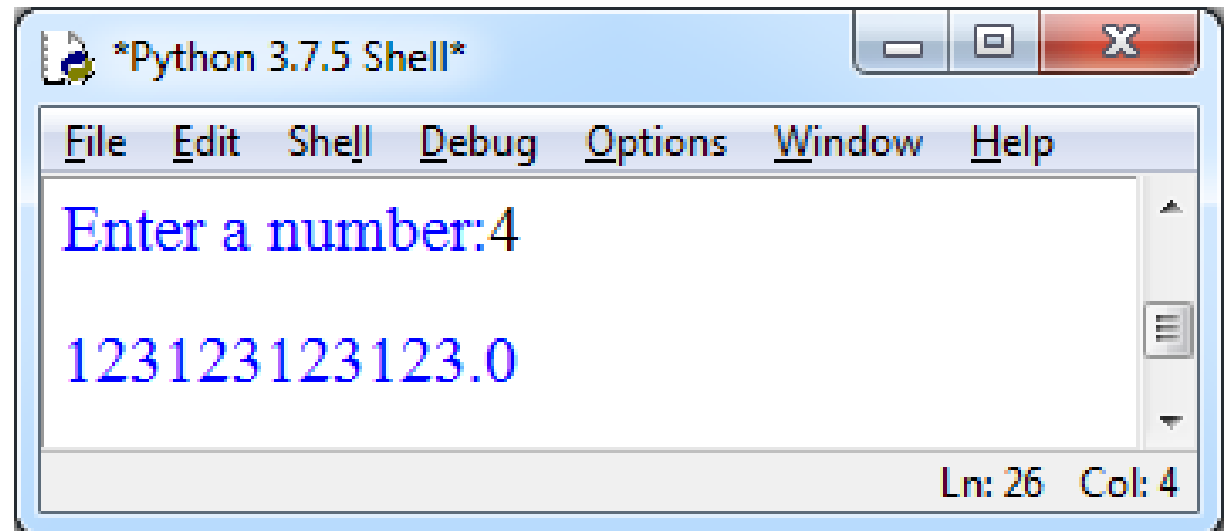
# ADD 2 STRINGS

```
a = '2'  
b = '3'  
print(a+b)
```



# MULTIPLY STRINGS

```
x = float('123' * int(input('Enter a number: ')))  
print(x)
```



# MULTIPLY STRINGS

```
x = float('123' * int(input('Enter a number: ')))  
print(x*2)
```

```
x = ('123' * int(input('Enter a number: ')))  
print(x*2)
```

```
Enter a number: 4  
246246246246.0  
Enter a number: 4  
123123123123123123123123
```

# MATH FUNCTIONS

Python has a math module that provides most of the familiar mathematical functions.

A **module** is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an **import statement**:

```
>>> import math
```

# MATH FUNCTIONS

This statement creates a **module object** named math. If you display the module object, you get some information about it:

```
>>> math
```

```
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

# MATH FUNCTIONS

```
>>> ratio = signal_power / noise_power  
>>> decibels = 10 * math.log10(ratio)  
>>> radians = 0.7  
>>> height = math.sin(radians)
```

The first example uses `math.log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined).



# MATH FUNCTIONS

```
>>> import math
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> print(math.sin(radians))
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the `math` module. Its value is a floating-point approximation of  $\pi$ , accurate to about 15 digits.

# Compute $\frac{\sqrt{2}}{2}$

If you know trigonometry, you can check the previous result by comparing it to the square root of 2 divided by 2:

```
>>> import math
```

```
>>> math.sqrt(2) / 2.0
```

```
0.707106781187
```

```
import math
```

```
n = float(input('n >> '))
```

```
print(math.sqrt(n))
```

# LOG FUNCTIONS IN PYTHON

**Syntax :** `math.log(a,Base)`

**Parameters :** **a :** The numeric value

**Base :** Base to which the logarithm has to be computed.

**Return Value :** Returns natural log if 1 argument is passed and log with specified base if 2 arguments are passed.

**Exceptions :** Raises `ValueError` if a negative no. is passed as argument

```
import math

# Printing the log base e of 14
print ("Natural logarithm of 14 is : ", end="")
print (math.log(14))

# Printing the log base 5 of 14
print ("Logarithm base 5 of 14 is : ", end="")
print (math.log(14,5))
```

Natural logarithm of 14 is : 2.6390573296152584

Logarithm base 5 of 14 is : 1.6397385131955606

# LOG FUNCTIONS IN PYTHON

**log2(a)** : This function is used to compute the **logarithm base 2** of a.

```
import math
# Printing the log base 2 of 14
print ("Logarithm base 2 of 14 is : ", end="")
print (math.log2(14))
```

**log10(a)** : This function is used to compute the **logarithm base 10** of a.


```
import math
# Printing the log base 10 of 14
print ("Logarithm base 10 of 14 is : ", end="")
print (math.log10(14))
```

# COMPOSITION

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
degrees = 75
```

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```



Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```
>>> minutes = hours * 60 # right
```

```
>>> hours * 60 = minutes # wrong!
```

```
SyntaxError: can't assign to operator
```

# ADDING NEW FUNCTIONS

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that run when the function is called.

Here is an example:

```
def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")
```

# ADDING NEW FUNCTIONS

```
def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")  
  
print_lyrics()
```



# ADDING NEW FUNCTIONS

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

# FUNCTIONS

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
>>> def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")
```

```
>>> def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()
```

```
>>> repeat_lyrics()  
I'm a boy, and I'm okay.  
I sleep all night and I work all day.  
I'm a boy, and I'm okay.  
I sleep all night and I work all day.
```

```
>>> def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")
```

```
>>> def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()
```

```
>>> repeat_lyrics()
```

# FUNCTIONS

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")  
  
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()  
  
repeat_lyrics()
```

# FUNCTIONS

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Now move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?



# FLOW OF EXECUTIONS

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called.

# PARAMETERS AND ARGUMENTS

Some of the functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the **arguments** are assigned to variables called **parameters**. Here is a definition for a function that takes an argument:

```
def print_twice(bruce):  
    print(bruce)  
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

# PARAMETERS AND ARGUMENTS

This function works with any value that can be printed:

```
>>> print_twice('Spam')
```

```
Spam
```

```
Spam
```

```
>>> print_twice(42)
```

```
42
```

```
42
```

```
>>> print_twice(math.pi)
```

```
3.14159265359
```

```
3.14159265359
```



# PARAMETERS AND ARGUMENTS

```
>>> print_twice('Spam '*4)
```

```
Spam Spam Spam Spam
```

```
Spam Spam Spam Spam
```

```
>>> print_twice(math.cos(math.pi))
```

```
-1.0
```

```
-1.0
```

# PARAMETERS AND ARGUMENTS

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
```

```
>>> print_twice(michael)
```

```
Eric, the half a bee.
```

```
Eric, the half a bee.
```

# PARAMETERS AND ARGUMENTS

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you assign the result to a variable, you get a special value called None:

```
>>> result = print_twice('Bing')
```

```
Bing
```

```
Bing
```

```
>>> print(result)
```

```
None
```

# LOCAL VARIABLES

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):  
    cat = part1 + part2  
    print_twice(cat)
```

# FUNCTIONS

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():  
    print("I'm a boy, and I'm okay.")  
    print("I sleep all night and I work all day.")  
  
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()  
  
repeat_lyrics()
```

# CAT\_TWICE()

```
import math
def print_twice(value):
    print(value)
    print(value)

def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)

cat_twice('CPE252 ', 'RSU')
cat_twice(2, 9)
print_twice('Spam')
print_twice(42)
print_twice(math.pi)
print_twice('Spam '*4)
```



Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '  
>>> line2 = 'tiddle bang.'  
>>> cat_twice(line1, line2)  
Bing tiddle tiddle bang.  
Bing tiddle tiddle bang.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

```
>>> print(cat)  
NameError: name 'cat' is not defined
```



# FRUITFUL FUNCTIONS AND VOID FUNCTIONS

Some of the functions we have used, such as the math functions, return results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.






# FRUITFUL FUNCTION

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.



# FRUITFUL FUNCTION (MATH.SQRT)

result; for example, you might assign it to a variable or use it as part of an expression:

```
golden = (math.sqrt(5) + 1) / 2
```

# RETURN VALUES

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
```

```
height = radius * math.sin(radians)
```



# RETURN VALUES

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

```
def area(radius):  
    a = math.pi * radius**2  
    return a
```

# RETURN VALUES

We have seen the return statement before, but in a fruitful function the return statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):  
    return math.pi * radius**2
```

# WHY FUNCTIONS?

There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

# FLOOR DIVISION AND MODULUS

The **floor division** operator, `//`, divides two numbers and rounds down to an integer.

For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105
```

```
>>> minutes / 60
```

```
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, dropping the fraction part:

```
>>> minutes = 105
```

```
>>> hours = minutes // 60
```

```
>>> hours
```

```
1
```

# FLOOR DIVISION AND MODULUS

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60
```

```
>>> remainder
```

```
45
```

An alternative is to use the **modulus operator**, %, which divides two numbers and returns the remainder:

```
>>> remainder = minutes % 60
```

```
>>> remainder
```

```
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if  $x \% y$  is zero, then  $x$  is divisible by  $y$ .



# FLOOR DIVISION AND MODULUS

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.

# SQUARE AREA

```
def areaOfSquare(s):  
    return s*s
```

```
s = float(input('s >> '))  
square_area = areaOfSquare(s)  
print(square_area)
```

# RECTANGLE AREA

```
def areaOfRectangle(width, length):  
    return width*length
```

```
width = float(input('Width >> '))  
length = float(input('Length >> '))
```

```
rectangle_area = areaOfRectangle (width, length);  
print(rectangle_area)
```

# DISTANCE BETWEEN TWO POINTS (FUNCTION)

```
import math
```

```
def distance2Points(x1, y1, x2, y2):  
    return math.sqrt((x2-x1)**2+(y2-y1)**2)
```

```
x1 = float(input('x1 >> '))
```

```
y1 = float(input('y1 >> '))
```

```
x2 = float(input('x2 >> '))
```

```
y2 = float(input('y2 >> '))
```

```
d = distance2Points(x1, y1, x2, y2)
```

```
print(d)
```

# LAST TWO AND THREE DIGITS PRIZES

```
import random

print('Last two digits prize: ',end='')
l2 = f'{random.randint(0, 9)}{random.randint(0, 9)}'
print(l2)

print('Last three digits prizes: ',end='')
f3_1 = f'{random.randint(0, 9)}{random.randint(0, 9)}{random.randint(0, 9)}'
f3_2 = f'{random.randint(0, 9)}{random.randint(0, 9)}{random.randint(0, 9)}'
print(f3_1, f3_2, sep='  ')
```

```
s = f'{5}{6}'
print(s)
print(f'{5}{6}')
a=5
b=6
s = f'{a}{b}'
print(s)
```

# LAST TWO AND THREE DIGITS PRIZES

```
print('Last three digits prizes: ',end='')  
print(random.randint(0, 9), random.randint(0, 9), random.randint(0, 9),  
      ' ', sep='', end='')  
print(random.randint(0, 9), random.randint(0, 9), random.randint(0, 9),  
      sep='')
```

Last two digits prize: 65

Last three digits prizes: 678 267

Last three digits prizes: 310 706

# PYTHON COMPARISON OPERATORS

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>&gt;</code>	Greater than	<code>x &gt; y</code>
<code>&lt;</code>	Less than	<code>x &lt; y</code>
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;= y</code>
<code>&lt;=</code>	Less than or equal to	<code>x &lt;= y</code>

# PYTHON LOGICAL OPERATORS

Operator	Description	Example
and	Returns True if both statements are true	<code>x &lt; 5 and x &lt; 10</code>
or	Returns True if one of the statements is true	<code>x &lt; 5 or x &lt; 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x &lt; 5 and x &lt; 10)</code>



# BOOLEAN EXPRESSIONS

The `==` operator is one of the **relational 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

# BOOLEAN EXPRESSIONS

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 relational operator. There is no such thing as =< or =>.

# 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:

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

```
True
```

```
>>> 5 == 6
```

```
False
```

# 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 or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

# LOGICAL OPERATORS

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`.

# 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 if is called the **condition**. If it is true, the indented statement runs. If not, nothing happens.

# CONDITIONAL EXECUTION

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          # TODO: need to handle negative values!
```

# CONDITIONAL EXECUTION

Python uses indentation to indicate a block of code.

## Example

```
if 5 > 2:  
    print("Five is greater than two!")
```



# CONDITIONAL EXECUTION (ERROR)

Python will give you an error if you skip the indentation:

## Example

Syntax Error:

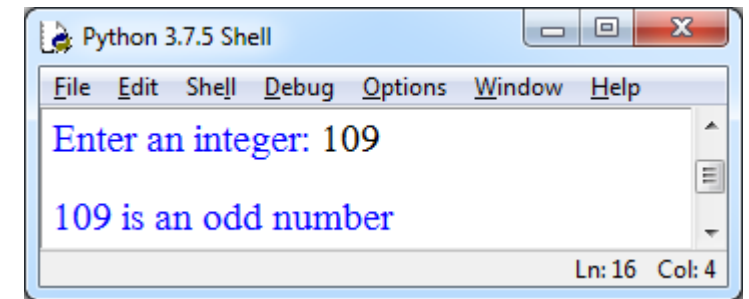
```
if 5 > 2:  
print("Five is greater than two!")
```

# CONDITIONAL EXECUTION

```
a = 33  
b = 200  
if b > a:  
    print("b is greater than a")
```

# ODD OR EVEN INTEGER NUMBER

```
n = int(input('Enter an integer: '))  
s = 'an odd number'  
if n % 2 == 0:  
    s = 'an even number'  
  
print(f'{n} is {s}')
```



```
n = int(input('Enter an integer: '))  
if n % 2 == 0:  
    s = 'an even number'  
else:  
    s = 'an odd number'  
  
print(f'{n} is {s}')
```

# ALTERNATIVE EXECUTION IF-ELSE

If the remainder when  $x$  is divided by 2 is 0, then we know that  $x$  is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs.

# ALTERNATIVE EXECUTION IF-ELSE

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

```
if x % 2 == 0:  
    print('x is even')  
else:  
    print('x is odd')
```

# ALTERNATIVE EXECUTION

```
a = 200
```

```
b = 33
```

```
if b > a:
```

```
    print("b is greater than a")
```

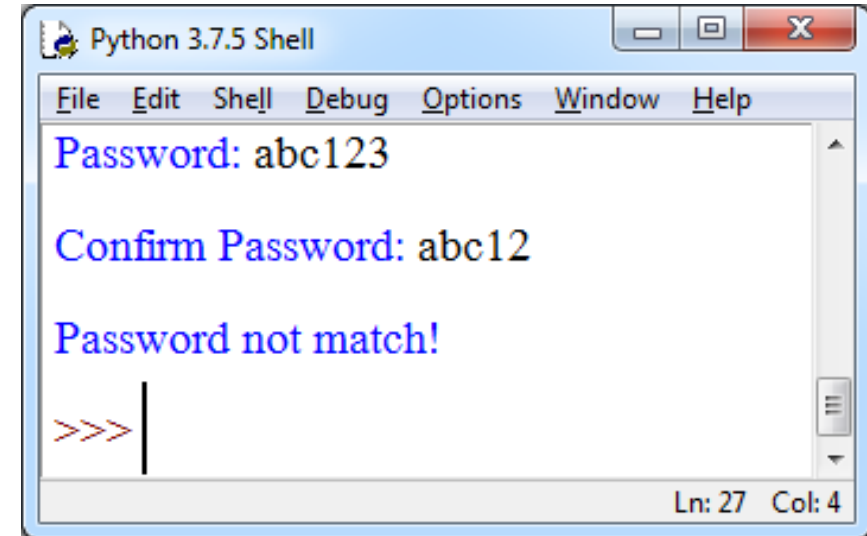
```
else:
```

```
    print("b is not greater than a")
```

# ALTERNATIVE EXECUTION IF-ELSE

```
pw1 = input('Password: ')
pw2 = input('Confirm Password: ')

if pw1 != pw2:
    print('Password not match!')
else:
    print('\nPassword is OK.')
```



# ALTERNATIVE EXECUTION IF-ELSE

```
pw1 = input('Password: ')
pw2 = input('Confirm Password: ')

if pw1 == pw2:
    print('Password is OK')
else:
    print('Password not match!')
```



# CHAINED CONDITIONALS

`elif` is an abbreviation of “else if ”. Again, exactly one branch will run. 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.

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("b is less than a")
```

# 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**:

```
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')
```



# CHAINED CONDITIONALS

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 runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

# AND

The and keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
```

```
b = 33
```

```
c = 500
```

```
if a > b and c > a:
```

```
    print("Both conditions are True")
```



# OR

The or keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
```

```
b = 33
```

```
c = 500
```

```
if a > b or a > c:
```

```
    print("At least one of the conditions is True")
```

# THE PASS STATEMENT

**if** statements cannot be empty, but if you for some reason have an **if** statement with no content, put in the **pass** statement to avoid getting an error.

```
a = 33  
b = 200
```

```
if b > a:  
    pass
```

# CHAINED CONDITIONALS

```
price = float(input('Enter a price: '))
if price < 5000:
    print(f'The price after discount : {price}.')
elif price >= 5000 and price < 10000:
    print(f'The price after discount : {0.9*price}.')
elif price >= 10000 and price < 50000:
    print(f'The price after discount : {0.8*price}.')
else:
    print(f'The price after discount : {0.7*price}.')
```

# CHAINED CONDITIONALS

```
price = float(input('Enter a price: '))  
if price < 5000:  
    print(f'The price after discount : {price}.')  
elif price < 10000:  
    print(f'The price after discount : {0.9*price}')  
elif price < 50000:  
    print(f'The price after discount : {0.8*price}')  
else:  
    print(f'The price after discount : {0.7*price}')
```



# NESTED CONDITIONALS

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

```
x=41
```

```
if x > 10:
```

```
    print("Above ten,")
```

```
    if x > 20:
```

```
        print("and also above 20!")
```

```
    else:
```

```
        print("but not above 20.")
```

```
x=41
```

```
if x > 10:
```

```
    print("Above ten,")
```

```
    if x > 20:
```

```
        print("and also above 20!")
```

```
    else:
```

```
        print("but not above 20.")
```

```
else:
```

```
    print("Not above ten")
```

# ROCK PAPER SCISSORS

```
player1 = int(input('Player #1: '))  
player2 = int(input('Player #2: '))  
result = ''
```

```
if player1 == 1:           #Rock  
    if player2 == 1:       #Rock  
        result = 'Tie'  
    elif player2 == 2:     #Paper  
        result = 'Player #2 wins'  
    elif player2 == 3:     #Scissors  
        result = 'Player #1 wins'
```

# ROCK PAPER SCISSORS

```
elif player1 == 2:  
    if player2 == 1:  
        result = 'Player #1 wins'  
    elif player2 == 2:  
        result = 'Tie'  
    elif player2 == 3:  
        result = 'Player #2 wins'
```

```
#Paper  
#Rock  
  
#Paper  
  
#Scissors
```

# ROCK PAPER SCISSORS

```
elif player1 == 3 :  
    if player2 == 1:  
        result= 'Player #2 wins'  
    elif player2 == 2 :  
        result = 'Player #1 wins'  
    elif player2 == 3:  
        result = 'Tie'
```

```
print(result)
```

```
#Scissors
```

```
#Rock
```

```
#Paper
```

```
#Scissors
```



# FIND THE SIZE OF THE POLO SHIRT

Find the size of the polo shirt according to the chest.

Less than 37 inches in size XS

From 37 but less than 41 inches in size S

From 41 but less than 43 inches in size M

From 43 but less than 46 inches in size L

From 46 inches onwards in size XL

# FIND THE SIZE OF THE POLO SHIRT

```
c = int(input())
if c < 37 :
    s = 'XS'
elif c < 41 :
    s = 'S'
elif c < 43 :
    s = 'M'
elif c < 46 :
    s = 'L'
else :
    s = 'XL'
print(s)
```

# A WAY TO SIMPLIFY NESTED CONDITIONAL STATEMENTS

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

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

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

```
if 0 < x and x < 10:  
    print('x is a positive single-digit number.')
```



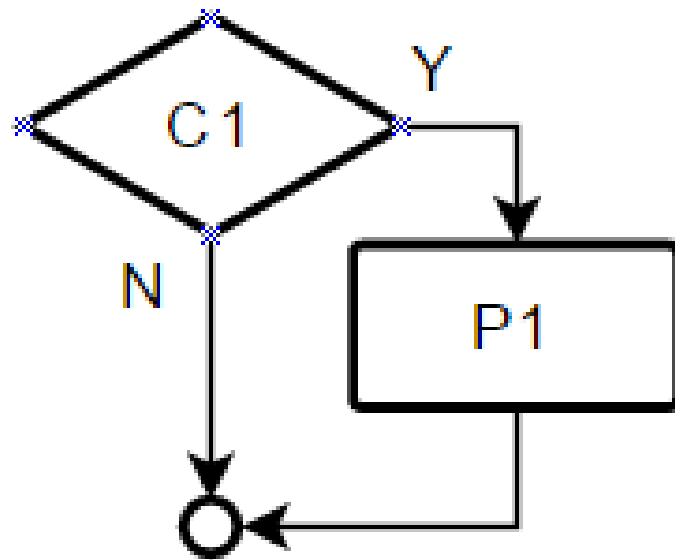
# IF

For this kind of condition, Python provides a more concise option:

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

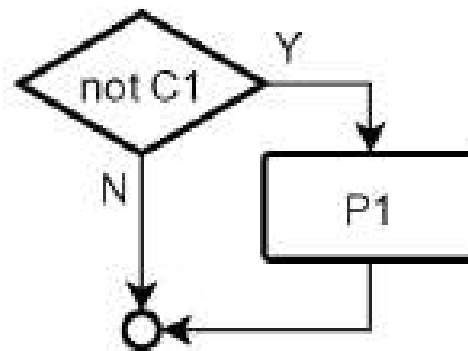
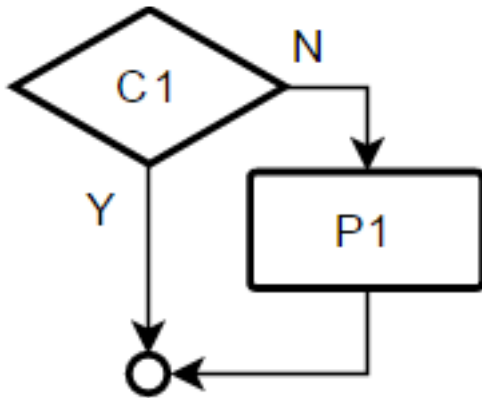


# SELECTION (IF)



if C1 :  
P1

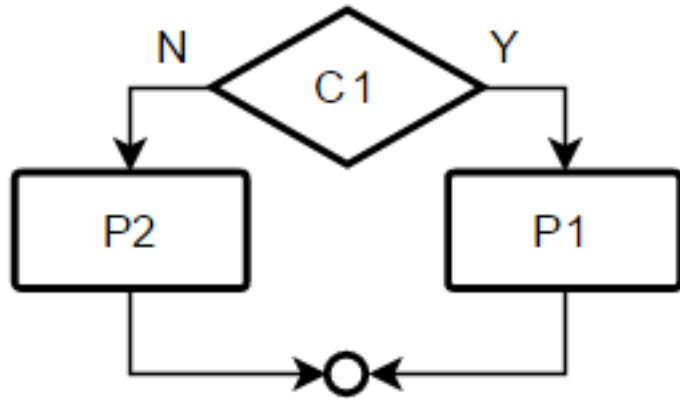
# SELECTION (IF)



if C1 :  
    pass  
else :  
    P1

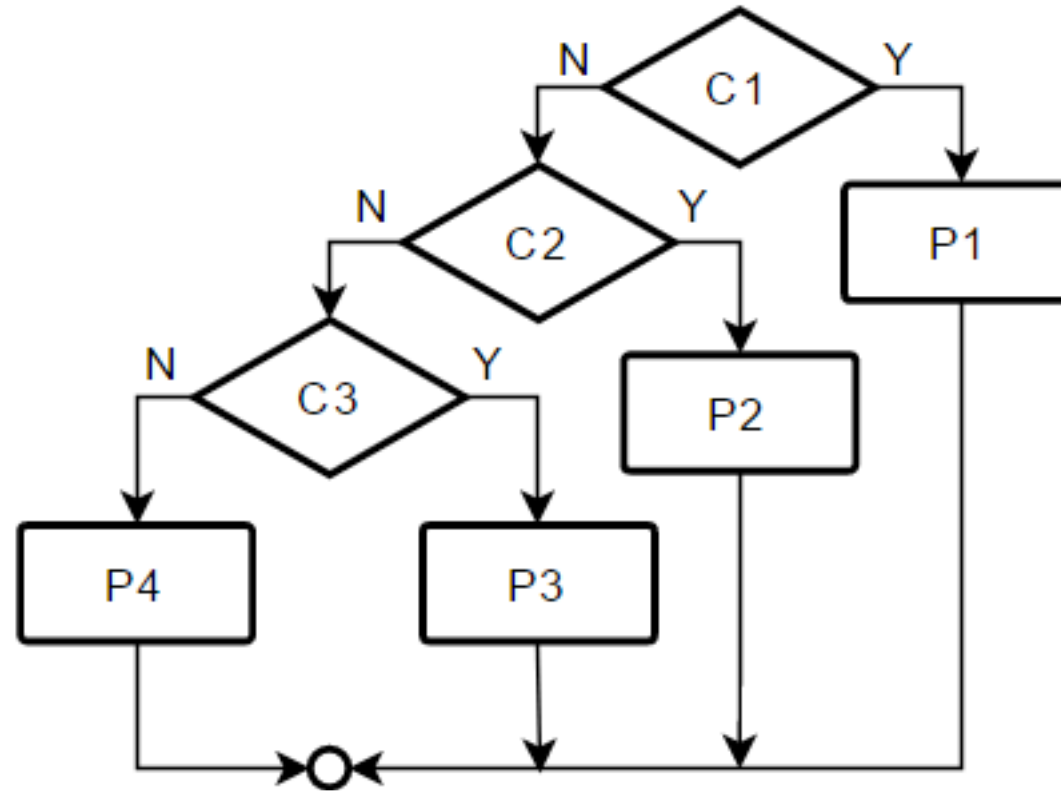
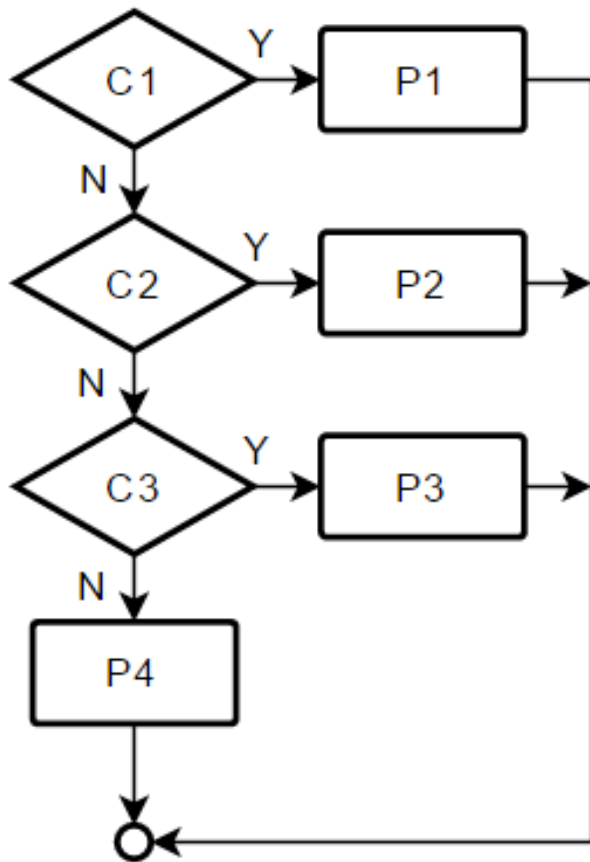
if not C1 :  
    P1

# SELECTION (IF)



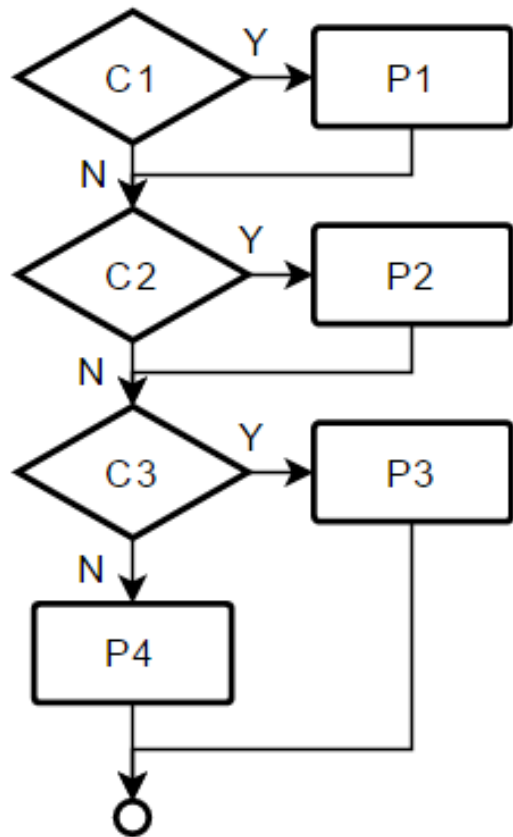
if C1 :  
    P1  
else :  
    P2

# SELECTION (IF)



if C1 :  
    P1  
elif C2 :  
    P2  
elif C3 :  
    P3  
else :  
    P4

# SELECTION (IF)



if C1 :

    P1

if C2 :

    P2

if C3 :

    P3


else :

    P4

MONEY  
(1000, 500, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.25)

```
import math

pay = float(input('Amount of money>> '))
remain = 0
b1000 = pay // 1000
remain = pay % 1000
b500 = remain // 500
remain %= 500
b100 = remain // 100
remain %= 100
b50 = remain // 50
remain %= 50
```



**MONEY**  
**(1000, 500, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.25)**

```
b20 = remain // 20  
remain %= 20  
b10 = remain // 10  
remain %= 10  
b5 = remain // 5  
remain %= 5  
b2 = remain // 2  
remain %= 2  
b1 = remain // 1  
remain %= 1
```

# MONEY

## (1000, 500, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.25)

```
b_50s = remain // 0.50
remain %= 0.50
b_25s = remain // 0.25
```

The `math.trunc()` method returns the truncated integer part of a number.

```
print(
    "B1000 = " + str(math.trunc(b1000)),
    "B500 = " + str(math.trunc(b500)),
    "B100 = " + str(math.trunc(b100)),
    "B50 = " + str(math.trunc(b50)),
```



MONEY  
(1000, 500, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.25)

```
"B20 = " + str(math.trunc(b20)),  
"B10 = " + str(math.trunc(b10)),  
"B5 = " + str(math.trunc(b5)),  
"B2 = " + str(math.trunc(b2)),  
"B1 = " + str(math.trunc(b1)),  
"B.50 = " + str(math.trunc(b_50s)),  
"B.25 = " + str(math.trunc(b_25s)),  
sep='\n'
```

)

# REPLACEABLE

<code>not(x == 0)</code>	<code>x != 0</code>
<code>not(x==2 or x==4)</code>	<code>x!=2 and x!=4</code>
<code>not(x&lt;2 and y&gt;=4)</code>	<code>x&gt;=2 or y&lt;4</code>
<code>3&lt;=x and x&lt;9</code>	<code>3 &lt;= x &lt; 9</code>
<code>a &lt; b and b &lt; c and c &lt; d and d &lt;= e</code>	<code>a &lt; b &lt; c &lt; d &lt;= e</code>
<code>c=='a' or c=='e' or c=='i' or c=='o' or c=='u'</code>	<code>c in ('a', 'e', 'i', 'o', 'u')</code> <code>c in 'aeiou'</code>

# KEYBOARD INPUT

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that stops the program 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.

```
>>> text = input()  
What are you waiting for?  
>>> text  
What are you waiting for?
```

# KEYBOARD INPUT

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. `input` can take a prompt as an argument:

```
>>> name = input('What...is your name?\n')
```


```
What...is your name?
```

```
Arthur, King of the Britons!
```

```
>>> name
```

```
Arthur, King of the Britons!
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.



If you expect the user to type an integer, you can try to convert the return value to int:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
```

```
>>> speed = input(prompt)
```

```
What...is the airspeed velocity of an unladen swallow?
```

```
42
```

```
>>> int(speed)
```

```
42
```



But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
```

What...is the airspeed velocity of an unladen swallow?

What do you mean, an African or a European swallow?

```
>>> int(speed)
```

ValueError: invalid literal for int() with base 10

We will see how to handle this kind of error later.



# DEBUGGING

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.



# DEBUGGING

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.





# DEBUGGING

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program and make small modifications, debugging them as you go.

# DEBUGGING

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was
- Where it occurred

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

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

```
>>> y = 6
```

**SyntaxError: unexpected indent**

# DEBUGGING

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

# DEBUGGING

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is  $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$ . In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

# DEBUGGING

When you run this program, you get an exception:

Traceback (most recent call last):

File "snr.py", line 5, in ?

decibels = 10 \* math.log10(ratio)

ValueError: math domain error

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of ratio, which turns out to be 0.

The problem is in line 4, which uses floor division instead of floating-point division.

# ABSOLUTE VALUE

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):
```

```
    if x < 0:
```

```
        return -x
```

```
    else:
```

```
        return x
```

```
x = float(input('x: '))
```

```
print(absolute_value(x))
```

Since these return statements are in an alternative conditional, only one runs.

# ABSOLUTE VALUE

```
def absolute_value(x):  
    if x < 0:  
        return -x  
    if x > 0:  
        return x
```

This function is incorrect because if  $x$  happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0:

```
>>> absolute_value(0)
```

```
None
```

# DISTANCE BETWEEN TWO POINTS

As an example, suppose you want to find the distance between two points, given by the coordinates  $x_1, y_1$  and  $x_2, y_2$  . By the Pythagorean theorem, the distance is:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



# DISTANCE BETWEEN TWO POINTS

```
import math
```

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx**2 + dy**2  
    result = math.sqrt(dsquared)  
    return result
```

# DISTANCE BETWEEN TWO POINTS (CONT.)

```
def distance(x1, y1, x2, y2):  
    return(math.sqrt((x2 - x1)**2 + (y2 - y1)**2))
```

```
x1 = float(input('x1 >> '))  
y1 = float(input('y1 >> '))  
x2 = float(input('x2 >> '))  
y2 = float(input('y2 >> '))
```

```
twoPointsDistance = distance(x1, y1, x2, y2)  
print(twoPointsDistance)
```

# THE RADIUS OF THE CIRCLE FROM 2 POINTS

The first step is to find the radius of the circle, which is the distance between the two points (a center and a point on circumference). We just wrote a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

# AREA OF A CIRCLE

พื้นที่วงกลม

```
def circle_area(xc, yc, xp, yp):  
    radius = distance(xc, yc, xp, yp)  
    result = math.pi*radius*radius  
    return result
```

```
import math

def distance(x1, y1, x2, y2):
    return(math.sqrt((x2 - x1)**2 + (y2 - y1)**2))

def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = math.pi*radius*radius
    return result
```

```
x1 = float(input('x1 >> '))  
y1 = float(input('y1 >> '))  
x2 = float(input('x2 >> '))  
y2 = float(input('y2 >> '))  
  
area = circle_area(x1, y1, x2, y2)  
print(area)
```

```
import math
x1 = float(input('x1 >> '))
y1 = float(input('y1 >> '))
x2 = float(input('x2 >> '))
y2 = float(input('y2 >> '))
radius = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
area = math.pi * radius * radius
print(radius)
print(area)
```

# BOOLEAN FUNCTIONS

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):  
    if x % y == 0:  
        return True  
    else:  
        return False
```

It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.





# BOOLEAN FUNCTIONS

Here is an example:

```
>>> is_divisible(6, 4)
```

```
False
```

```
>>> is_divisible(6, 3)
```

```
True
```

# BOOLEAN FUNCTIONS

The result of the `==` operator is a boolean, so we can write the function more concisely by returning it directly:

```
def is_divisible(x, y):  
    return x % y == 0
```

# BOOLEAN FUNCTIONS

Boolean functions are often used in conditional statements:

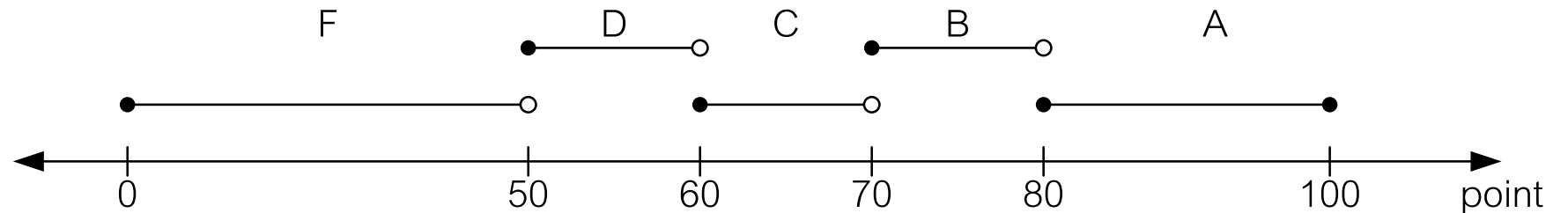
```
if is_divisible(x, y):  
    print('x is divisible by y')
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:  
    print('x is divisible by y')
```

# GRADE FROM A SCORE

```
score = float(input('score >> '))
if score >= 0 and score <=100 :
    if score < 50:
        print('F')
    elif score < 60:
        print('D')
    elif score < 70:
        print('C')
    elif score < 80:
        print('B')
    else:
        print('A')
else:
    print('Incorrect input')
```



# GRADE FROM A SCORE

```
score = float(input('score >> '))
if 0<=score<=100 :
    if 0<=score < 50:
        print('F')
    elif 50<=score < 60:
        print('D')
    elif 60<=score < 70:
        print('C')
    elif 70<=score < 80:
        print('B')
    else:
        print('A')
else:
    print('Incorrect input')
```

# UPDATING VARIABLES

A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

```
>>> x = x + 1
```

This means “get the current value of `x`, add one, and then update `x` with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> x = x + 1
```

```
NameError: name 'x' is not defined
```

# ITERATION (UPDATING VARIABLES)

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
```

```
>>> x = x + 1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.



# THE WHILE STATEMENT

More formally, here is the flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.



# THE WHILE STATEMENT

```
def countdown(n):  
    while n > 0:  
        print(n)  
        n = n - 1  
    print('Blastoff!')
```

You can almost read the while statement as if it were English. It means, “While  $n$  is greater than 0, display the value of  $n$  and then decrement  $n$ . When you get to 0, display the word Blastoff!”

# THE WHILE STATEMENT

```
def countdown(n):  
    while n > 0:  
        print(n)  
        n = n - 1  
    print('Blastoff!')  
  
countdown(10)
```

10

Blastoff!

9

Blastoff!

8

Blastoff!

7

Blastoff!

6

Blastoff!

5

Blastoff!

4

Blastoff!

3

Blastoff!

2

Blastoff!

1

Blastoff!



# THE WHILE STATEMENT

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

# BREAK

Sometimes you don't know it's time to end a loop until you get halfway through the body. In that case you can use the break statement to jump out of the loop. For example, suppose you want to take input from the user until they type done. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

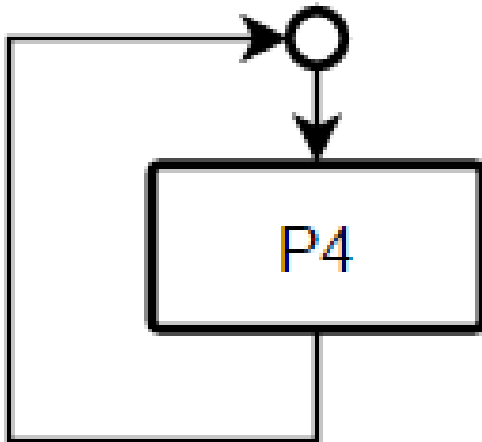
# BREAK

The loop condition is True, which is always true, so the loop runs until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

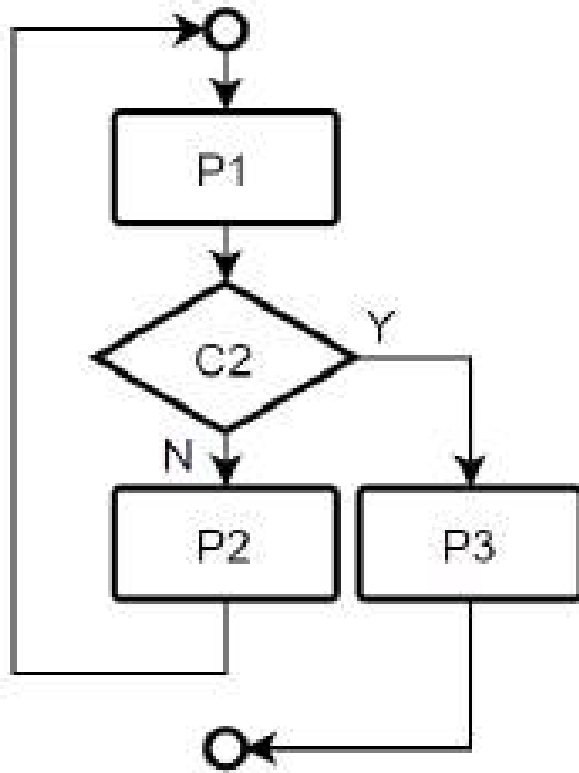
```
> not done  
not done  
> done  
Done!
```

# THE WHILE STATEMENT



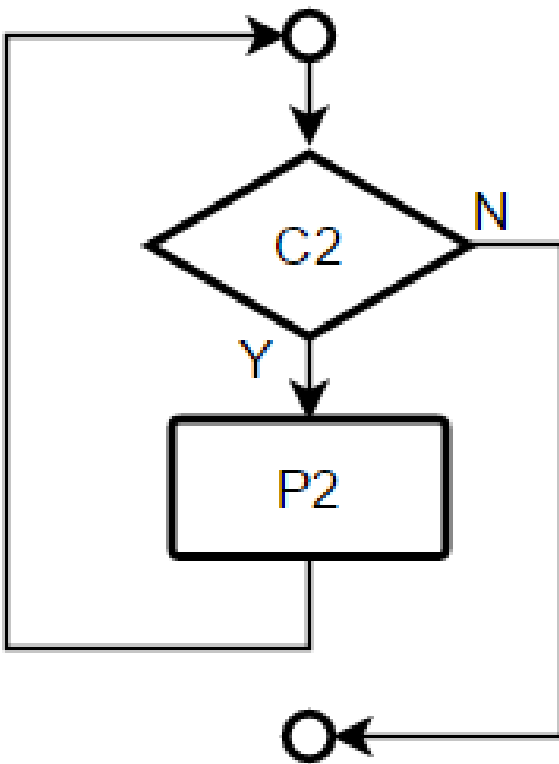
```
while True :  
    P4
```

# THE WHILE STATEMENT



```
while True :  
    P1  
    if C2 :  
        P3  
        break  
    P2
```

# THE WHILE STATEMENT



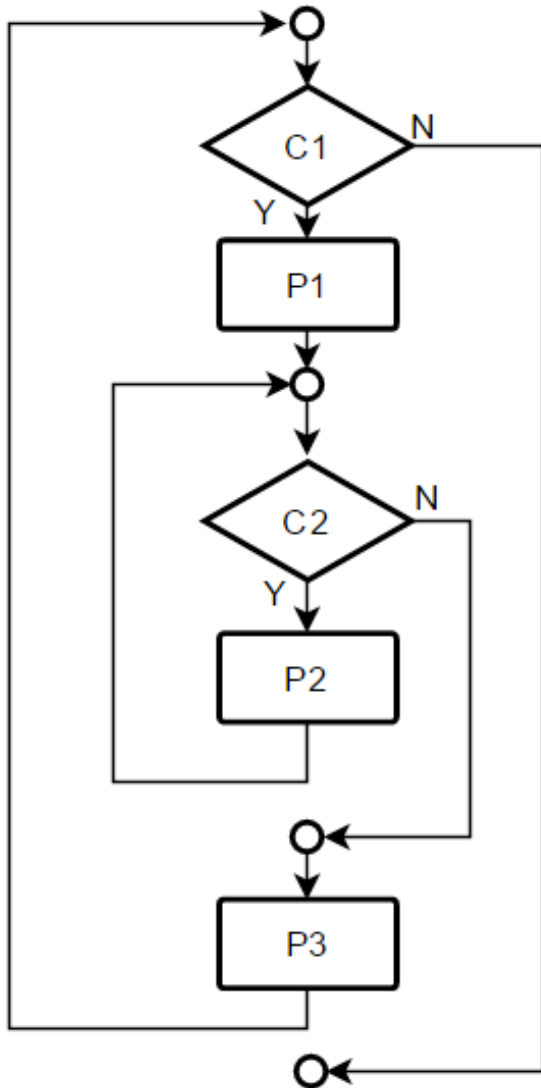
```
while True :  
    if not C2 : break  
    P2
```

```
while C2 :  
    P2
```

```
while True :  
    if not C2 :  
        break  
    P2
```

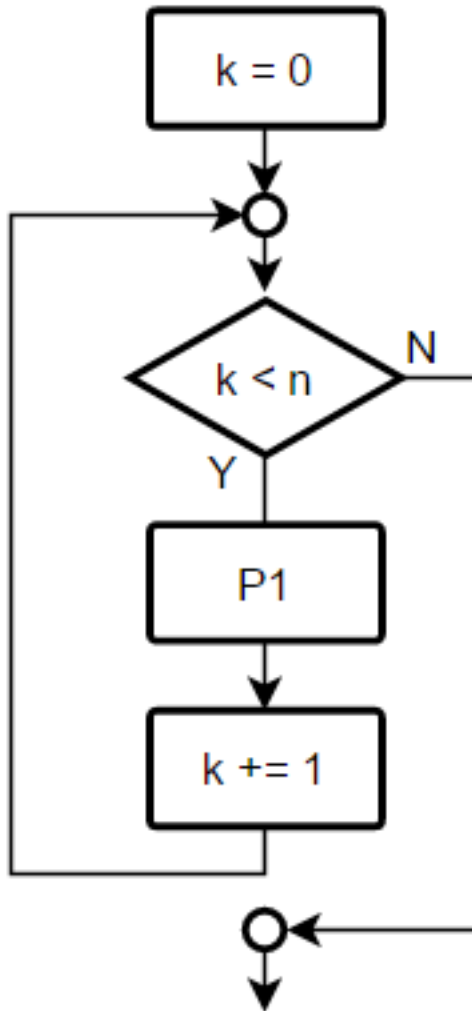


# THE WHILE STATEMENT



```
while C1 :  
    P1  
    while C2 :  
        P2  
    P3
```

# THE WHILE STATEMENT



$k = 0$   
while  $k < n$  :  
     $P1$   
     $k += 1$

# RECEIVE A PASSWORD

```
validCode = False

while not validCode:
    code = input('Enter a password>>')
    if code == '1234':
        validCode = True

print('Correct password')
```

# THE FOR STATEMENT

- for k in range(10) : k = 0, 1, 2, ..., 9
- for k in range(2,10) : k = 2, 3, 4, ..., 9
- for k in range(2,10,2) : k = 2, 4, 6, 8
- for k in range(10,1,-2) : k = 10, 8, 6, 4, 2
- for k in range(11,11) : Not execute any statement (step is positive and start >= stop)
- for k in range(9,10,-1) : Not execute any statement (step is negative and start <= stop)



# THE FOR STATEMENT

```
for i in range(4):  
    print('Hello!')
```

You should see something like this:

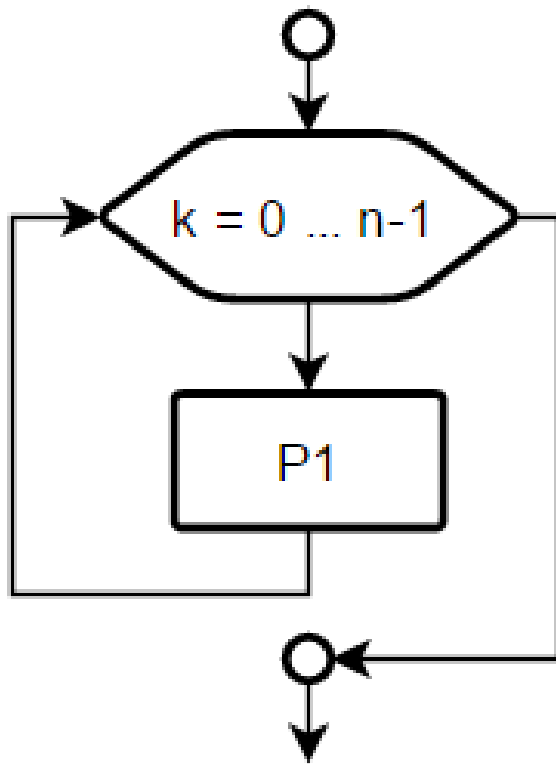
Hello!

Hello!

Hello!

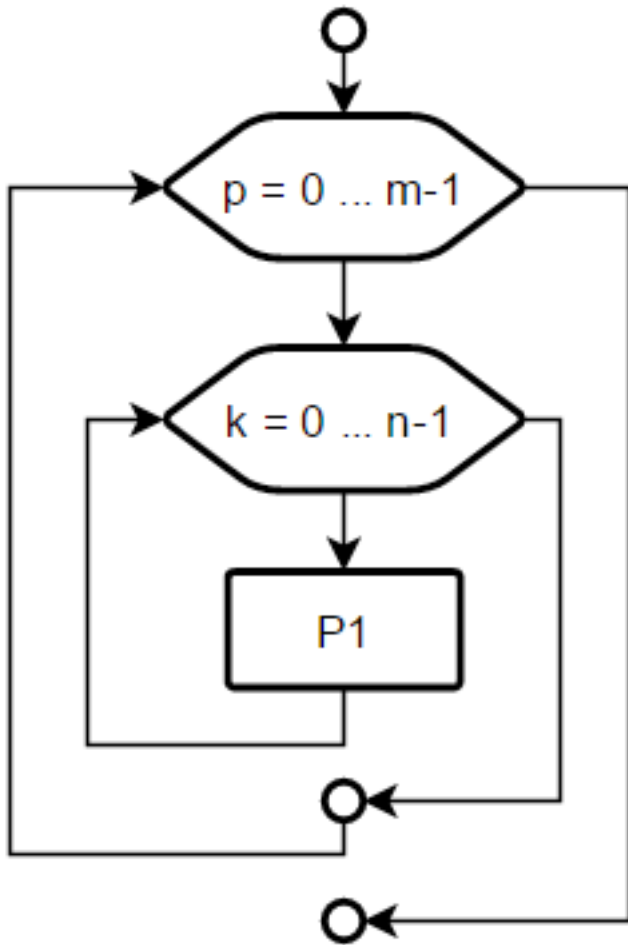
Hello!

# THE FOR STATEMENT



for k in range(n) :  
    P1

# THE FOR STATEMENT



```
for p in range(m) :  
    for k in range(n) :  
        P1
```

```
for p in range(4) :  
    for k in range(3) :  
        print(p, k)
```

0 0

0 1

0 2

1 0

1 1

1 2

2 0

2 1

2 2

3 0

3 1

3 2

# FOR LOOP

# Prints out the numbers 0,1,2,3,4 (different lines)

```
for x in range(5):  
    print(x)
```

# Prints out 3,4,5 (different lines)

```
for x in range(3, 6):  
    print(x)
```

# Prints out 3,5,7 (different lines)

```
for x in range(3, 8, 2):  
    print(x)
```



# WHILE LOOP

# Prints out 0,1,2,3,4 (different lines)

```
count = 0
```

```
while count < 5:
```

```
    print(count)
```

```
    count += 1 # This is the same as count = count + 1
```

```
for i in range(1,6):    # 1,2,3,4,5
    for j in range(i):
        print('x', end="")
    print("")
```

x

xx

xxx

xxxx

xxxxx

```
for i in range(4, 0,-1):  
    for j in range(i):  
        print('x', end="")  
    print("")
```

XXXX

XXX

XX

X

```
for i in range(1,5):  
    for j in range(i):  
        print('x', end="")  
    print("")  
for i in range(3, 0,-1):  
    for j in range(i):  
        print('x', end="")  
    print("")
```

```
x  
xx  
xxx  
xxxx  
xxx  
xx  
x
```

```
x=3
```

```
y=2
```

```
for i in range(x):
```

```
    for j in range(y):
```

```
        print("Rangsit")
```

Rangsit

Rangsit

Rangsit

Rangsit

Rangsit

Rangsit

# DISPLAY ODD NUMBERS FROM 1 TO 999

แสดงตัวเลขจำนวนคี่ตั้งแต่ 1 ถึง 999

```
for x in range(1000): #0,1..., 999
    # Check if x is even
    if x % 2 == 0:
        continue
    print(x)
```

```
for i in range(11):  
    if i%2==0 :  
        continue  
    else:  
        print(i)
```

1

3

5


7

9

```
for i in range(11):  
    if i%2==0 :  
        continue  
    else:  
        print(i**3)
```

1  
27  
125  
343  
729







```
for i in range(11):  
    if i%2 !=0 :  
        print(i**3)
```

```
i = 0  
while i <=10:  
    if i%2!=0 :  
        print(i**3)  
    i +=1
```


1  
27  
125  
343  
729


$$\sum_{i=1}^{100} i$$

```
sum = 0  
for i in range(1,101):  
    sum = sum+i  
  
print(sum)
```


$$\sum_{i=1}^{200} i$$

```
sum = 0  
for i in range(1,201):  
    sum += i  
print(sum)
```


$$\sum_{i=1}^n i$$

```
n = int(input('n >> '))  
sum = 0  
for i in range(1,n+1):  
    sum += i  
print(sum)
```

$$\sum_{i=1}^n \frac{1}{i^2}$$

```
n = int(input('n >> '))  
sum = 0  
for i in range(1,n+1):  
    sum += 1/(i*i)  
print(sum)
```

# BREAK AND CONTINUE

# Prints out 0,1,2,3,4 (different lines)

```
count = 0
```

```
while True:
```

```
    print(count)
```

```
    count += 1
```

```
    if count >= 5:
```

```
        break
```

# Prints out only odd numbers 1,3,5,7,9 (different lines)

```
for x in range(10):
```

```
    # Check if x is even
```

```
    if x % 2 == 0:
```

```
        continue
```

```
    print(x)
```

```
money = 10
for year in range(2,6):
    pay = year*money
    print("Year ",year,"Money ",pay)
    continue
if(year==3):
    print("Additional payment 1000 equals", pay+1000)
```

Year 2 Money 20

Year 3 Money 30

Year 4 Money 40

Year 5 Money 50



```
money = 10
for year in range(2,6):
    pay = year*money
    print("Year ",year,"Money ",pay)
    if(year==3):
        print("Additional payment 1000 equals", pay+1000)
        break
```

Year 2 Money 20

Year 3 Money 30

Additional payment 1000 equals 1030

# AVERAGE OF 10 NUMBERS

```
s = 0
for k in range(10) :
    a = float(input())
    s += a
print('average =', (s/10))
```

# PRIME NUMBERS

```
q = int(input('q= '))
for k in range(2, q+1) : # k = 2,3,...,q
    if q % k == 0 : break
if k == q :
    print(q,'is prime')
else:
    print(q, '=', k, 'x', q//k)
    print(q,'is not prime')
```

q= 47

47 is prime

q= 57

57 = 3 x 19

57 is not prime

# PRIME NUMBERS

```
q = int(input('q= '))  
for k in range(2, q+1) : # k = 2,3,...,q  
    if q % k == 0 :  
        break
```

```
if k == q :  
    print(q, ' is prime')  
else:  
    print(q, '=', k, 'x', q//k)  
    print(q, ' is not prime')
```

```
q= 1  
Traceback (most recent call last):  
  File "C:/Program Files/Python37/prime.py", line 6, in  
<module>  
    if k == q :  
NameError: name 'k' is not defined
```

# PRIME NUMBERS

```
q = int(input('q= '))
for k in range(2, q+1): # k = 2,3,...,q
    if q % k == 0 :
        break
if q == 1:
    print('q = 1 is not prime')
elif k == q :
    print(q, ' is prime')
else:
    print(q, '=', k, 'x', q//k)
    print(q, ' is not prime')
```

# STRINGS

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
```

```
>>> letter = fruit[1]
```

# STRINGS

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> letter
```

```
'a'
```

# STRINGS

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
```

```
>>> letter
```

```
'b'
```



# STRINGS

As an index, you can use an expression that contains variables and operators:

```
>>> i = 1
```

```
>>> fruit[i]
```

```
'a'
```

```
>>> fruit[i+1]
```

```
'n'
```

# STRINGS

```
s1 = 'I am "Python".'  
s2= 'I\'m Python.'  
s3="I'm Python."  
s4 ="I am \"Python\"."  
s5="'"I'm "Python"'"  
s6=""""I'm "Python".""""
```

# LEN

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
```

```
>>> len(fruit)
```

```
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
```

```
>>> last = fruit[length]
```

```
IndexError: string index out of range
```

# STRINGS

The reason for the `IndexError` is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

# TRAVERSAL WITH A LOOP

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```



# TRAVERSAL WITH A LOOP

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when index is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

# TRAVERSAL WITH A LOOP

Another way to write a traversal is with a for loop:

```
for letter in fruit:  
    print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable letter. The loop continues until no characters are left.

# CONCATENATION (STRING ADDITION)

The following example shows how to use concatenation (string addition) and a for Loop.

```
prefixes = 'JKLMNOPQ'
```

```
suffix = 'ack'
```

```
for letter in prefixes:
```

```
    print(letter + suffix)
```

**The output is:**

Jack

Kack

Lack

Mack

Nack

Oack

Pack

Qack



# STRING SLICES

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
```

```
>>> s[0:5]
```

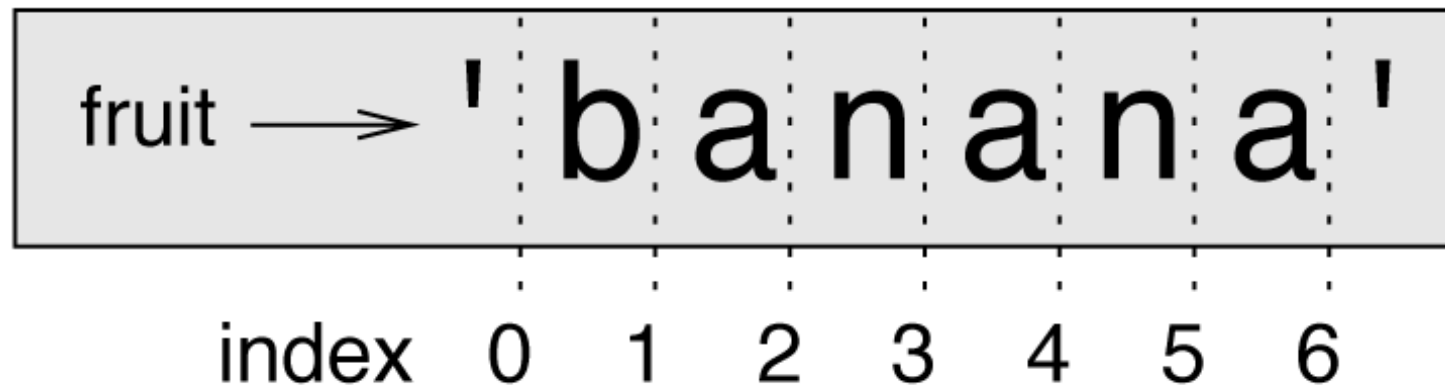
```
'Monty'
```

```
>>> s[6:12]
```

```
'Python'
```

# STRING SLICES

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in the following figure.



# STRING SLICES

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
```

```
>>> fruit[:3]
```

```
'ban'
```

```
>>> fruit[3:]
```

```
'ana'
```

# STRING SLICES

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
```

```
>>> fruit[3:3]
```

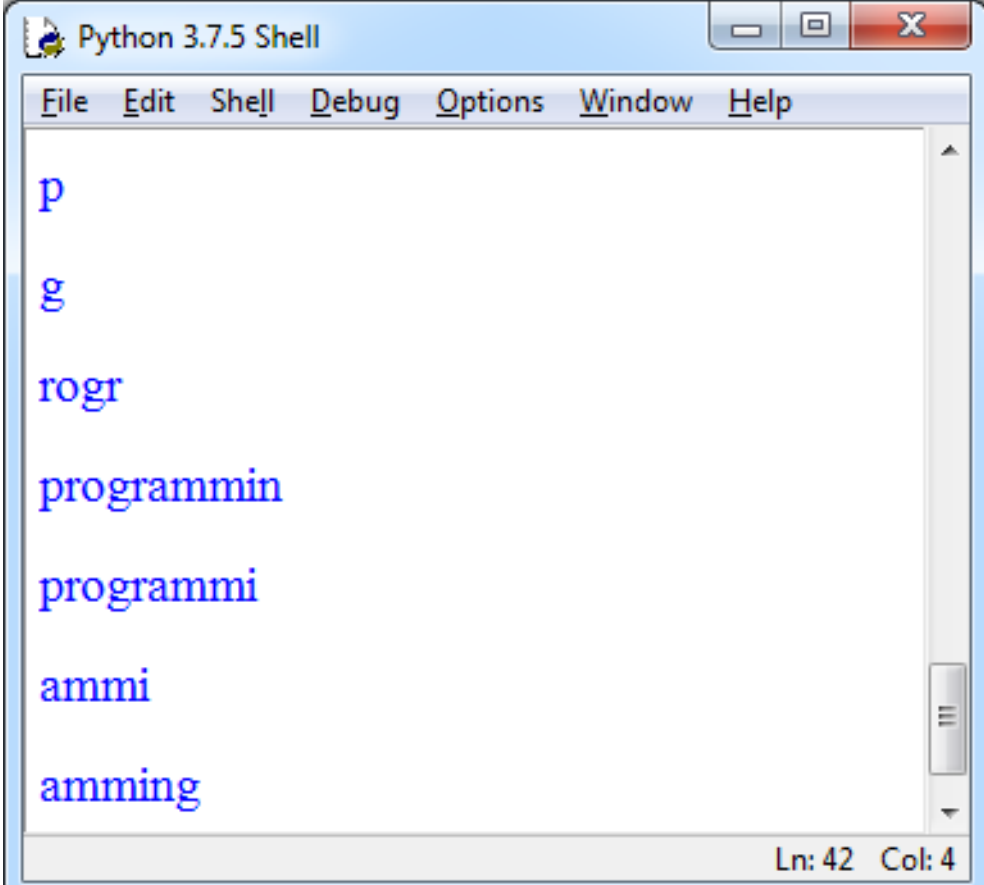
```
''
```

# ACCESSING CHARACTERS BY POSITIVE AND NEGATIVE INDEX NUMBER

p	r	o	g	r	a	m	m	i	n	g
0	1	2	3	4	5	6	7	8	9	10
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

# ACCESSING CHARACTERS BY POSITIVE AND NEGATIVE INDEX NUMBER

```
str = 'programming'  
print(str[0])  
print(str[-1])  
print(str[1:5])  
print(str[0:-1])  
print(str[0:-2])  
print(str[5:-2])  
print(str[5:])
```



The screenshot shows a Python 3.7.5 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help) and a text area containing the output of the code. The output is as follows:

```
p  
g  
rogr  
programmin  
programmi  
ammi  
amming
```

The status bar at the bottom right indicates "Ln: 42 Col: 4".

# STRINGS ARE IMMUTABLE

- It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
```

```
>>> greeting[0] = 'J'
```

```
TypeError: 'str' object does not support item assignment
```

# STRINGS ARE IMMUTABLE

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.



# SEARCHING (A CHARACTER)

```
def find(word, letter):  
    index = 0  
    while index < len(word):  
        if word[index] == letter:  
            return index  
        index = index + 1  
    return -1
```

In a sense, find is the inverse of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

```
def find(word, letter):  
    index = 0  
    while index < len(word):  
        if word[index] == letter:  
            return index  
        index = index + 1  
    return -1  
print(find('banana', 'a'))  
print(find('banana', 'x'))  
pos=find('Pineapple', 'a')  
print(pos)
```

# SEARCHING (A CHARACTER)

This is the first example we have seen of a return statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately. If the character doesn't appear in the string, the program exits the loop normally and returns -1.

# LOOPING AND COUNTING

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

# STRING METHODS

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters. Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`:

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

```
word = 'bAnAnA'  
new_word = word.upper()  
print(new_word)  
new_word = word.lower()  
print(new_word)
```

# STRING METHODS

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no arguments.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

# STRING METHODS

Actually, the find method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
```

```
2
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
```

```
4
```



# STRING METHODS

This is an example of an **optional argument**. `find` can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range.

```
name = 'bob'  
pos=name.find('b', 1, 2)  
print(pos)  
name = 'bananas'  
pos=name.find('s', 1,7)  
print(pos)
```

# THE IN OPERATOR

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
```

```
True
```

```
>>> 'seed' in 'banana'
```

```
False
```

```
a= 'a' in 'banana'
```

```
b= 'seed' in 'banana'
```

```
print(a)
```

```
print(b)
```

# STRING COMPARISON

The relational operators work on strings. To see if two strings are equal:


```
if word == 'banana':  
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':  
    print('Your word, ' + word + ', comes before banana.')
```

```
elif word > 'banana':  
    print('Your word, ' + word + ', comes after banana.')
```

```
else:  
    print('All right, bananas.')
```



```
word = input().lower()
print(word)
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```



# STRING COMPARISON

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

# ARRAY

Diagram illustrating the components of an array declaration:

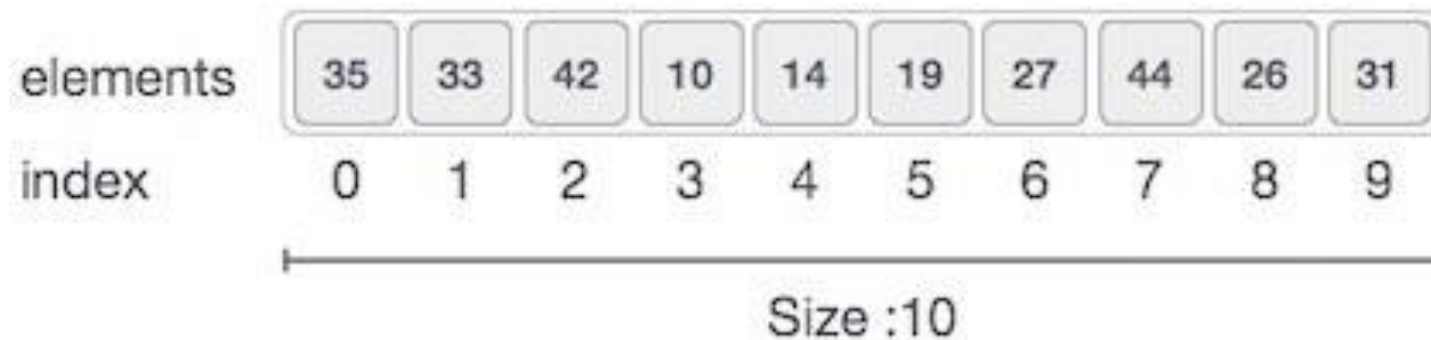
```
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
```

Labels and arrows:

- Name:** Points to `array`
- Elements:** Points to the list of values: `{ 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }`
- Type:** Points to `int`
- Size:** Points to `[10]`

## Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.







# ARRAY

- As per the above illustration, following are the important points to be considered.
- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index.

# ARRAY

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *  
arrayName = array(typecode, [Initializers])
```

# ARRAY

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
l	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

# ACCESSING ARRAY

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
for x in array1:  
    print(x)
```

Output

10  
20  
30  
40  
50

# ACCESSING ARRAY ELEMENT

```
from array import *  
array1 = array('i', [10, 20, 30, 40, 50])  
print (array1[0])  
print (array1[2])
```

Output

```
10  
30
```

```
from array import *  
array1 = array('d', [10.5,20.9,30.4,40.8,50.6])  
for i in range(5):  
    print(array1[i])
```

# INSERTION OPERATION

```
from array import *  
array1 = array('i', [10, 20, 30, 40, 50])  
array1.insert(1, 60)  
for x in array1:  
    print(x)
```

Output

10  
60  
20  
30  
40  
50

# DELETION OPERATION

- Deletion refers to removing an existing element from the array and re-organizing all elements of an array.
- Here, we remove a data element at the middle of the array using the python in-built `remove()` method.

```
from array import *  
array1 = array('i', [10, 20, 30, 40, 50])  
array1.remove(40)  
for x in array1:  
    print(x)
```

## Output

```
10  
20  
30  
50
```



# SEARCH OPERATION

- You can perform a search for an array element based on its value or its index.
- Here, we search a data element using the python in-built index() method.

```
from array import *  
array1 = array('i', [10, 20, 30, 40, 50])  
print (array1.index(40))
```

Output

3

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

# UPDATE OPERATION

- Update operation refers to updating an existing element from the array at a given index.
- Here, we simply reassign a new value to the desired index we want to update.

```
from array import *  
array1 = array('i', [10, 20, 30, 40, 50])  
array1[2] = 80  
for x in array1:  
    print(x)
```

## Output

```
10  
20  
80  
40  
50
```

# FILES

To write a file, you have to open it with mode 'w' as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

open returns a file object that provides methods for working with the file. The write method puts data into the file:

```
>>> line1 = "This here's the wattle,\n">>> fout.write(line1)>>> fout.close()
```

```
fout = open('output.txt', 'w')  
line1 = "This here's the wattle,\n"  
fout.write(line1)  
fout.close()
```

# WRITE A FILE

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
>>> x = 52
```

```
>>> fout.write(str(x))
```

# READ A LINE IN A TEXT FILE

```
fin = open('input.txt')  
line = fin.readline()  
print(line)  
fin.close()
```

# READ & WRITE TEXT FILES

```
fin = open('input.txt')  
fout = open('output.txt', 'w')
```

```
text = fin.readline()  
while text:  
    text = text.rstrip('\n')  
    print(text, file = fout)  
    text = fin.readline()  
fout.close()  
fin.close()
```

# FORMAT OPERATOR

An alternative is to use the **format operator**, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator. The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42. A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

# FORMAT OPERATOR

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')  
'In 3 years I have spotted 0.1 camels.'
```



# FORMAT OPERATOR

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
```

```
TypeError: not enough arguments for format string
```

```
>>> '%d' % 'dollars'
```

```
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

# TRY-EXCEPTION

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an IOError:

```
>>> fin = open('bad_file')  
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')  
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

# TRY-EXCEPTION

The idea of the *try-except block* is this:

- **try**: the code with the exception(s) to catch. If an exception is raised, it jumps straight into the except block.
- **except**: this code is only executed *if an exception occurred* in the try block. The except block is required with a try block, even if it contains only the pass statement.

It may be combined with the **else** and **finally** keywords.

- **else**: Code in the else block is only executed if no exceptions were raised in the try block.
- **finally**: The code in the finally block is always executed, regardless of if a an exception was raised or not.

# TRY-EXCEPTION

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the try statement does. The syntax is similar to an if...else statement:

```
try:  
    fin = open('bad_file')  
except:  
    print('Something went wrong.')
```

Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and runs the except clause.

# TRY-EXCEPTION

try:

```
    fin = open('input.txt')
```

```
    for line in fin:
```

```
        print(line)
```

```
    fin.close()
```

except:

```
    print('Something went wrong.')
```

# TRY-EXCEPTION

try:

```
radius=float(input('Radius: '))
```

```
circle_area=22/7*radius*radius
```

```
print(f'Radius of the circle is {radius} and area of the circle is {circle_area:.2f} Sq m.')
```

except:

```
print('Failed to get a radius (Exception occurred)')
```

else:

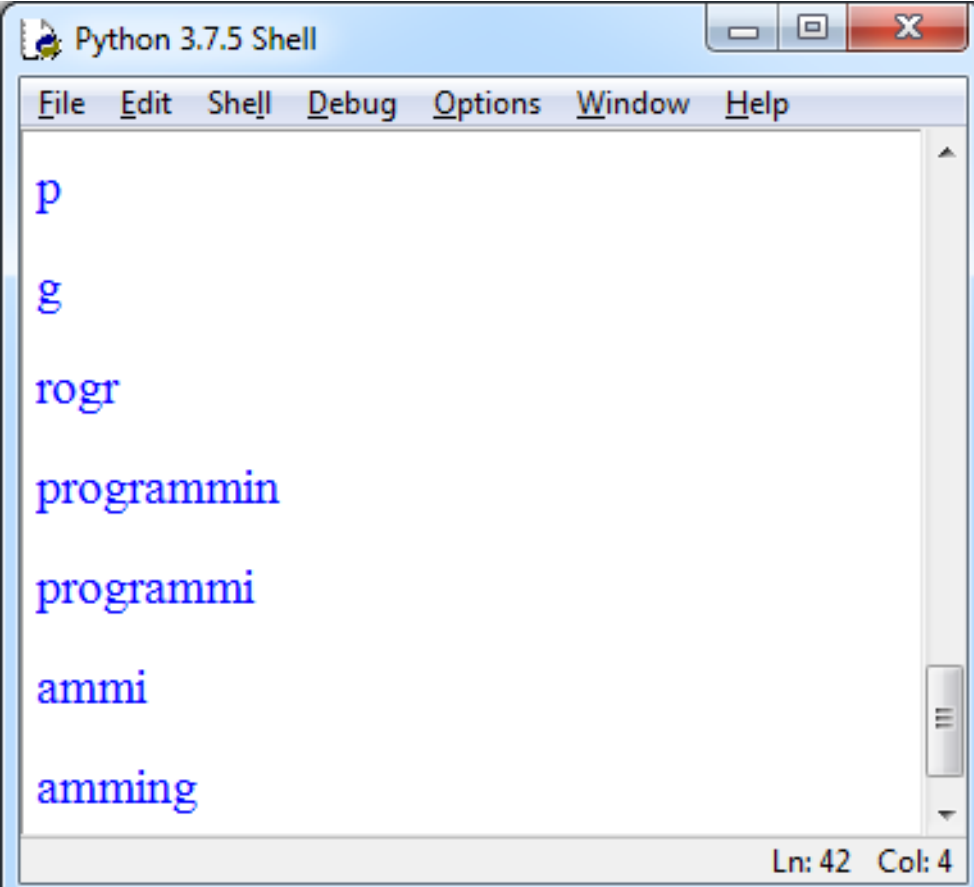
```
print('No exception occurred')
```

finally:

```
print('We always do this')
```

# ACCESSING CHARACTERS BY POSITIVE AND NEGATIVE INDEX NUMBER

```
str = 'programming'  
print(str[0])  
print(str[-1])  
print(str[1:5])  
print(str[0:-1])  
print(str[0:-2])  
print(str[5:-2])  
print(str[5:])
```



The screenshot shows a Python 3.7.5 Shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help) and a text area containing the output of the code. The output is as follows:

```
p  
g  
rogr  
programmin  
programmi  
ammi  
amming
```

The status bar at the bottom right indicates "Ln: 42 Col: 4".

# STRINGS ARE IMMUTABLE

- It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
```

```
>>> greeting[0] = 'J'
```

```
TypeError: 'str' object does not support item assignment
```



# STRINGS ARE IMMUTABLE

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

# LISTS

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**. There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

```
[10, 20, 30, 40]
```

```
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type.

# A LIST IS A SEQUENCE

The following list contains a string, a float, an integer, and another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

# A LIST IS A SEQUENCE

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
```

```
>>> numbers = [42, 123]
```

```
>>> empty = []
```

```
>>> print(cheeses, numbers, empty)
```

```
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

```
cheeses = ['Cheddar', 'Edam', 'Gouda']  
numbers = [42, 123]  
empty = []  
print(cheeses, numbers, empty)  
print(f'{cheeses} {numbers} {empty}')
```

# LISTS ARE MUTABLE

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> cheeses[0]  
'Cheddar'
```

```
cheeses = ['Cheddar', 'Edam', 'Gouda']  
numbers = [42, 123]  
empty = []  
print(f'{cheeses[1]} {numbers[1]} {empty}')
```

# LISTS ARE MUTABLE

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned:

```
>>> numbers = [42, 123]
```

```
>>> numbers[1] = 5
```

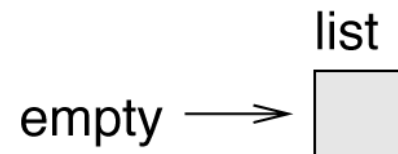
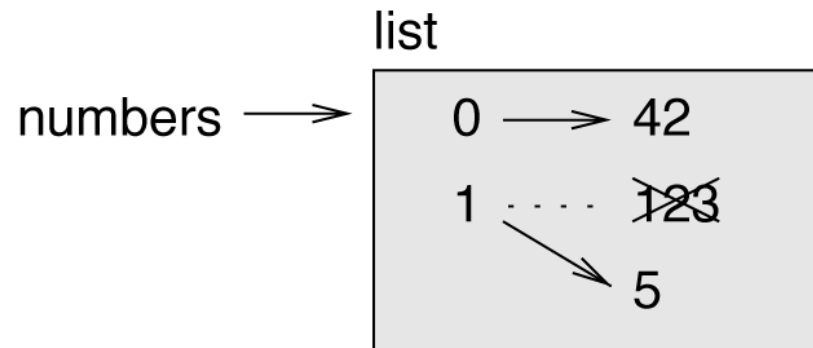
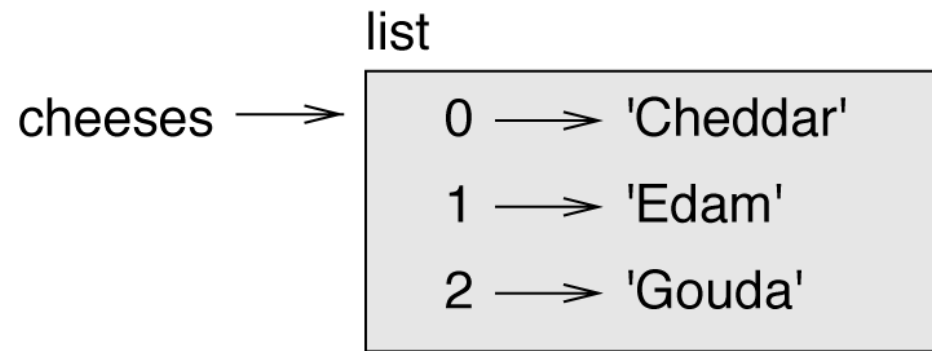
```
>>> numbers
```

```
[42, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.



# THE STATE DIAGRAM FOR CHEESES, NUMBERS AND EMPTY



List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

# IN OPERATOR

The in operator also works on lists:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
```

```
>>> 'Edam' in cheeses
```

```
True
```

```
>>> 'Brie' in cheeses
```

```
False
```

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

```
print('Edam' in cheeses)
```

```
if 'Edam' in cheeses :  
    print('Edam likes cheeses')
```

# TRAVERSING A LIST

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for cheese in cheeses:  
    print(cheese)
```

# TRAVERSING A LIST

การเข้าถึงค่าต่างๆที่อยู่ในลิสต์

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to  $n-1$ , where  $n$  is the length of the list. Each time through the loop, `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

```
numbers = [1,2,3,4,5]
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
print(numbers)
```

```
def change_all(numbers):  
    for i in range(len(numbers)):  
        numbers[i] = numbers[i] * 2  
    return numbers
```

```
t=[1,2,3]  
print(change_all(t))  
print(t)
```

[2, 4, 6]

[2, 4, 6]

# TRAVERSING A LIST

A for loop over an empty list never runs the body:

```
for x in []:  
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['RSU', 'CPE', 'Mail'], [1, 2, 3]]
```



# LIST OPERATIONS (+ OPERATOR)

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
```

```
>>> b = [4, 5, 6]
```

```
>>> c = a + b
```

```
>>> c
```

```
[1, 2, 3, 4, 5, 6]
```

```
a = [1, 2, 3]
```

```
b = [4, 5, 6]
```

```
c = a + b
```

```
d=[a[0]+b[0], a[1]+b[1]]
```

```
print(c)
```

```
print(d)
```

# \* OPERATOR

```
>>> [0] * 4
```

```
[0, 0, 0, 0]
```

```
>>> [1, 2, 3] * 3
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

```
a=[0] * 4
```

```
print(a)
```

```
b=[1, 2, 3] * 3
```

```
print(b)
```

# LIST SLICES

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
print(t[1:3])
print(t[:4])
print(t[3:])
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list:

# LIST SLICES

```
>>> t[:]  
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> t[1:3] = ['x', 'y']  
>>> t  
['a', 'x', 'y', 'd', 'e', 'f']
```

# LIST METHODS

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
```

```
>>> t.append('d')
```

```
>>> t
```

```
['a', 'b', 'c', 'd']
```

# LIST METHODS

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
```

```
>>> t2 = ['d', 'e']
```

```
>>> t1.extend(t2)
```

```
>>> t1
```

```
['a', 'b', 'c', 'd', 'e']
```

# LIST METHODS

This example leaves t2 unmodified.

**sort** arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

```
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
u = ['pineapple', 'banana', 'ant']
u.sort()
print(u)
```

Most list methods are void; they modify the list and return None. If you accidentally write `t = t.sort()`, you will be disappointed with the result.



```
nums=[11, 2, 33, 4, 5, 6]
nums.sort()      # [2, 4, 5, 6, 11, 33]
print(nums[2], nums[-3])  #5 6
for item in nums:    # [2, 4, 5, 6, 11, 33]
    if(item%2 !=0):
        print("Hello")
```

5 6

Hello

Hello

Hello

# SUM NUMBERS IN A LIST

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):  
    total = 0  
    for x in t:  
        total = total+x  
    return total
```

```
t = [1, 2, 3, 4, 5]  
print(add_all(t))
```

15



To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):  
    total = 0  
    for x in t:  
        total += 3*x  
    return total
```

```
t = [1, 2, 3, 4, 5]  
print(add_all(t))
```

45

# sum()

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, `sum`:

```
>>> t = [1, 2, 3]
```

```
>>> sum(t)
```

```
6
```

```
def a(x):  
    for i in range(len(x)):  
        x[i] *= 3  
    return sum(x)
```

# x = [3, 6, 9, 12]  
# 3+6+9+12 = 30

```
k = [1, 2, 3, 4]
```

```
print(a(k))
```

```
print(k)
```

# 30

# [3, 6, 9, 12]

30

[3, 6, 9, 12]

```
def b(x):  
    x *= 3  
    return x
```

```
def c(x):  
    x = [4*e for e in x]  
    return sum(x)
```

```
a = 2  
print(b(a))  
print(a)  
y = [1, 2, 3]  
print(c(y))
```

6

2

24



```
def b(x):
```

```
    x *= 3
```

```
    return x
```

```
def c(x):
```

```
    x = [3*e for e in x]
```

```
    return sum(x)
```

```
a = 2
```

```
b(a)
```

```
print(a)
```

```
y = [1, 2, 3]
```

```
print(c(y))
```

```
print(y)
```

2

18

[1, 2, 3]

# CONVERT TO A CAPITALIZED LIST

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):  
    res = []  
    for s in t:  
        res.append(s.capitalize())  
    return res
```

```
print(capitalize_all("abc"))
```

`['A', 'B', 'C']`



# CONVERT TO A CAPITALIZED LIST

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it “maps” a function (in this case the method `capitalize`) onto each of the elements in a sequence.

# KEEP ONLY UPPER CHARACTERS

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):  
    res = []  
    for s in t:  
        if s.isupper():  
            res.append(s)  
    return res
```

isupper is a string method that returns True if the string contains only uppercase letters.

An operation like only\_upper is called a **filter** because it selects some of the elements and filters out the others.

# DELETING ELEMENTS

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
```

```
>>> x = t.pop(1)
```

```
>>> t
```

```
['a', 'c']
```

```
>>> x
```

```
'b'
```

**pop** modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

you can use the del operator:

```
>>> t = ['a', 'b', 'c']
```

```
>>> del t[1]
```

```
>>> t
```

```
['a', 'c']
```

# REMOVE

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']  
>>> t.remove('b')  
>>> t  
['a', 'c']
```



```
s = [1, 2, 1, 3, 1, 5, 6]
```

```
x = 1
```

```
while x in s:
```

```
    s.remove(x)
```

```
print(s)
```

```
[2, 3, 5, 6]
```

# DEL

To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> del t[1:5]
```

```
>>> t
```

```
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.



# STRING TO LIST

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
```

```
>>> t = list(s)
```

```
>>> t
```

```
['s', 'p', 'a', 'm']
```

# SPLIT()

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
>>> s = 'pining for the fjords'  
>>> t = s.split()  
>>> t  
['pining', 'for', 'the', 'fjords']
```

# LISTS AND STRINGS

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

# LISTS AND STRINGS

**join** is the inverse of **split**. It takes a list of strings and concatenates the elements. **join** is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pinning', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pinning for the fjords'
```

In this case the delimiter is a space character, so **join** puts a space between words. To concatenate strings without spaces, you can use the empty string, "", as a delimiter.

# TUPLE

```
x = (10, 20, 30, 40, 50, 'c', 'd', 'e')
```

```
x[0] = 6
```

```
x[2] = 7
```

```
x[4] = 8
```

```
print(x)
```

Traceback (most recent call last):

File "C:\Program Files\Python37\ex1.py", line 2, in  
<module>

x[0] = 6

TypeError: 'tuple' object does not support item assignment

(10, 20, 30, 40, 50, 'c', 'd', 'e')

# LIST

```
x = [10, 20, 30, 40, 50, 'c', 'd', 'e']
```

```
x[0] = 'a'
```

```
x[5] = 7
```

```
x[6] = 8
```

```
print(x)
```

```
['a', 20, 30, 40, 50, 7, 8, 'e']
```

# DIFFERENCE BETWEEN LIST AND TUPLE

## LIST

Lists are mutable

Implication of iterations is time-consuming

The list is better for performing operations, such as insertion and deletion.

Lists consume more memory

Lists have several built-in methods

The unexpected changes and errors are more likely to occur

## TUPLE

Tuples are immutable

The implication of iterations is comparatively faster

Tuple data type is appropriate for accessing the elements

Tuple consume less memory as compared to the list

Tuple does not have many built-in methods.

In tuple, it is hard to take place

# TUPLE

```
x = (10, 20, 30, 40, 50, 'c', 'd', 'e')
```

```
print(x)
```

```
print(x[1], x[3])
```

```
x = (10, 20, 30, 40, 50)
```

```
print(x)
```

(10, 20, 30, 40, 50, 'c', 'd', 'e')

20 40

(10, 20, 30, 40, 50)



# RECURSION

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):  
    if n <= 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n-1)
```

# RECURSION

If  $n$  is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs  $n$  and then calls a function named `countdown`—itself—passing  $n-1$  as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

# RECURSION

The execution of countdown begins with  $n=3$ , and since  $n$  is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with  $n=2$ , and since  $n$  is greater than 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with  $n=1$ , and since  $n$  is greater than 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with  $n=0$ , and since  $n$  is not greater than 0, it outputs the word, "Blastoff!" and then returns.

The countdown that got  $n=1$  returns.

The countdown that got  $n=2$  returns.

The countdown that got  $n=3$  returns.



# RECURSION

And then you're back in `__main__`. So, the total output looks like this:

3

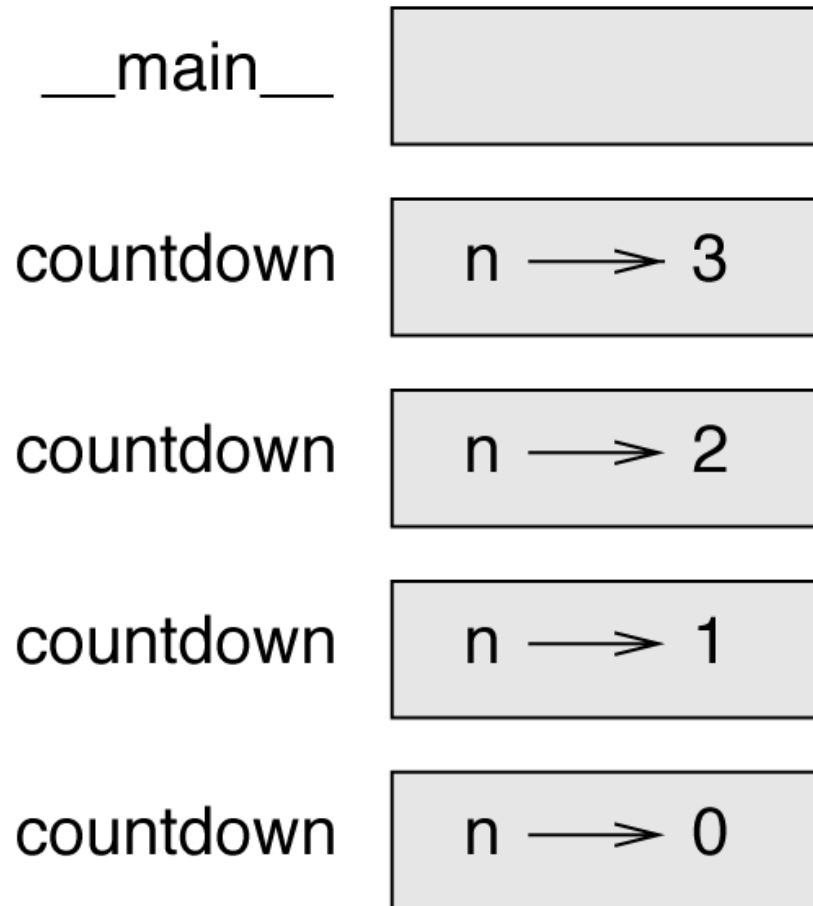
2

1

Blastoff!

A function that calls itself is **recursive**; the process of executing it is called **recursion**.

# STACK DIAGRAM



# RECURSION

As another example, we can write a function that prints a string n times:

```
def print_n(s, n):
```

```
    if n <= 0:
```

```
        return
```

```
    print(s)
```

```
    print_n(s, n-1)
```

```
print_n("RSU Engineers", 5)
```

RSU Engineers

RSU Engineers

RSU Engineers

RSU Engineers

RSU Engineers

# RECURSION

If  $n \leq 0$  the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to countdown: it displays  $s$  and then calls itself to display  $s$   $n-1$  additional times. So the number of lines of output is  $1 + (n - 1)$ , which adds up to  $n$ .

# RECURSION

```
def print_n(s, n):  
    if n <= 0:  
        return  
    print(s, n)  
    print_n(s, n-1)  
  
print_n("RSU Engineers", 5)
```

RSU Engineers 5

RSU Engineers 4

RSU Engineers 3

RSU Engineers 2

RSU Engineers 1





# RECURSION

For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.



# INFINITE RECURSION

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():  
    recurse()
```

# INFINITE RECURSION

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
File "<stdin>", line 2, in recurse
```

```
File "<stdin>", line 2, in recurse
```

```
File "<stdin>", line 2, in recurse
```

```
.
```

```
.
```

```
.
```

```
File "<stdin>", line 2, in recurse
```

```
RuntimeError: Maximum recursion depth exceeded
```

# FACTORIAL

$$0! = 1$$

$$n! = n(n - 1)!$$

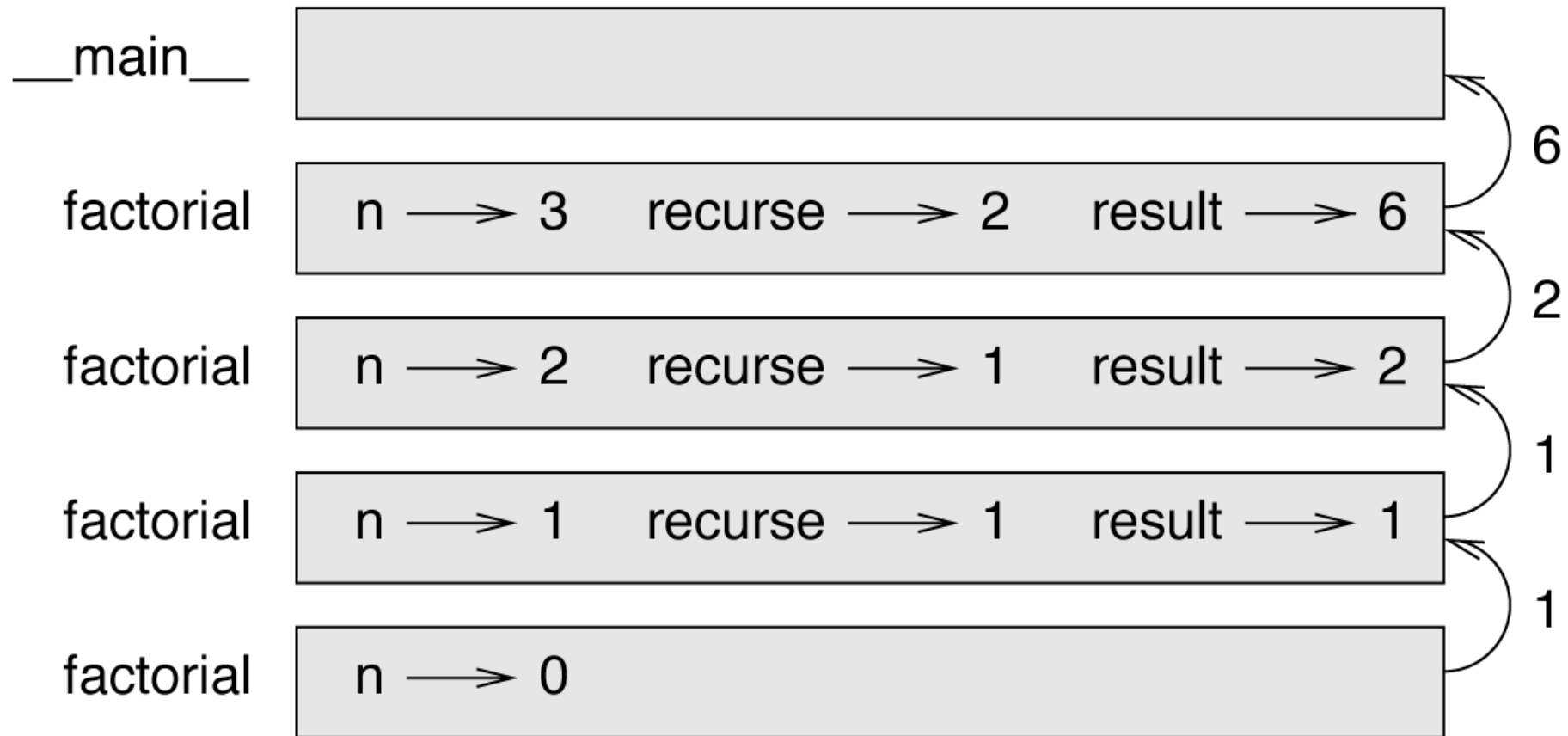
This definition says that the factorial of 0 is 1, and the factorial of any other value,  $n$ , is  $n$  multiplied by the factorial of  $n-1$ .

So  $3!$  is 3 times  $2!$ , which is 2 times  $1!$ , which is 1 times  $0!$ . Putting it all together,  $3!$  equals 3 times 2 times 1 times 1, which is 6.

# FACTORIAL

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        recurse = factorial(n-1)  
        result = n * recurse  
        return result
```

# STACK DIAGRAM



# CHECKING TYPES

What happens if we call factorial and give it 1.5 as an argument?

```
>>> factorial(1.5)
```

```
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case—when  $n == 0$ . But if  $n$  is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of  $n$  is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

# FACTORIAL

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial (n):  
    if not isinstance(n, int):  
        print('Factorial is only defined for integers.')  
        return None  
    elif n < 0:  
        print('Factorial is not defined for negative integers.')  
        return None  
    elif n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```



# FACTORIAL

```
>>> factorial('fred')
```

Factorial is only defined for integers.

None

```
>>> factorial(-2)
```

Factorial is not defined for negative integers.

None

If we get past both checks, we know that  $n$  is positive or zero, so we can prove that the recursion terminates.

# FIBONACCI

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

# FIBONACCI

```
def fibonacci (n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

# CASE STUDY: INTERFACE DESIGN (THE TURTLE MODULE)

```
import turtle  
bob = turtle.Turtle()  
print(bob)  
turtle.mainloop()
```

The turtle module (with a lowercase *t*) provides a function called Turtle (with an uppercase *T*) that creates a Turtle object, which we assign to a variable named bob.

This means that bob refers to an object with type Turtle as defined in module turtle.

mainloop tells the window to wait for the user to do something, although in this case there's not much for the user to do except close the window.

# CASE STUDY: INTERFACE DESIGN (THE TURTLE MODULE)

Once you create a Turtle, you can call a **method** to move it around the window. A method is similar to a function, but it uses slightly different syntax. For example, to move the turtle forward:

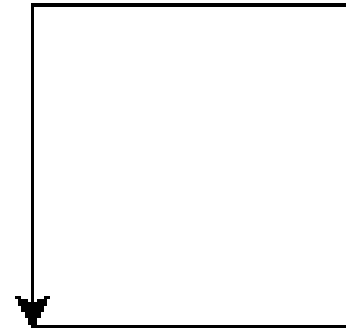
```
bob.fd(100)
```

The method, fd, is associated with the turtle object we're calling bob. Calling a method is like making a request: you are asking bob to move forward.

The argument of fd is a distance in pixels, so the actual size depends on your display. Other methods you can call on a Turtle are bk to move backward, lt for left turn, and rt right turn. The argument for lt and rt is an angle in degrees.

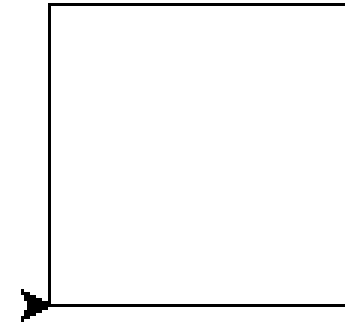
# SIMPLE REPETITION

```
import turtle
bob = turtle.Turtle()
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
```



# SIMPLE REPETITION

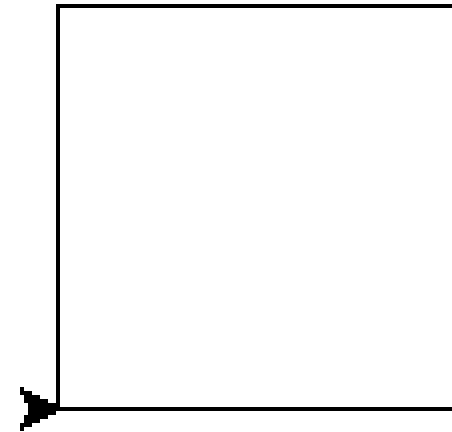
```
import turtle
bob = turtle.Turtle()
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```



# USE A FUNCTION

```
import turtle  
bob = turtle.Turtle()
```

```
def square(t):  
    for i in range(4):  
        t.fd(100)  
        t.lt(90)  
square(bob)
```





# USE A FUNCTION

```
import turtle
bob = turtle.Turtle()

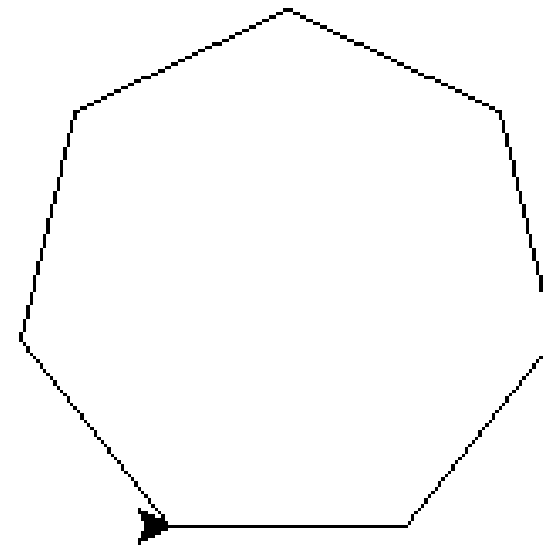
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
square(bob, 100)
```

# POLYGON

```
import turtle
bob = turtle.Turtle()

def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```



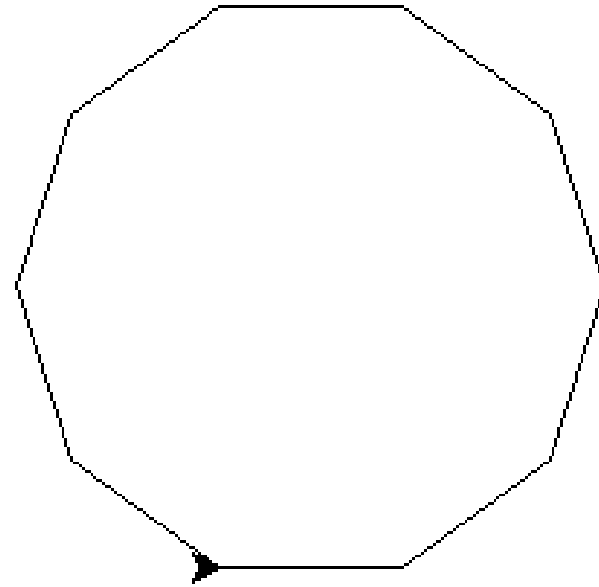
# POLYGON

```
import math
import turtle
```

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 10
    length = circumference / n
    polygon(t, n, length)
```

```
bob = turtle.Turtle()
circle(bob, 100)
```



# TKINTER PROGRAMMING

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications.

- Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps –
- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

# PYTHON TKINTER – MESSAGEBOX WIDGET

Python Tkinter – MessageBox Widget is used to display the message boxes in the python applications. This module is used to display a message using provides a number of functions.

Syntax:

`messagebox.Function_Name(title, message [, options])`

- **Function\_Name:** This parameter is used to represents an appropriate message box function.
- **title:** This parameter is a string which is shown as a title of a message box.
- **message:** This parameter is the string to be displayed as a message on the message box.
- **options:** There are two options that can be used are:
  - **default:** This option is used to specify the default button like ABORT, RETRY, or IGNORE in the message box.
  - **parent:** This option is used to specify the window on top of which the message box is to be displayed.

# MESSAGEBOX

- **Function\_Name:**  
There are functions or methods available in the messagebox widget.
- **showinfo():** Show some relevant information to the user.
- **showwarning():** Display the warning to the user.
- **showerror():** Display the error message to the user.
- **askquestion():** Ask question and user has to answered in yes or no.
- **askokcancel():** Confirm the user's action regarding some application activity.
- **askyesno():** User can answer in yes or no for some action.
- **askretrycancel():** Ask the user about doing a particular task again or not.

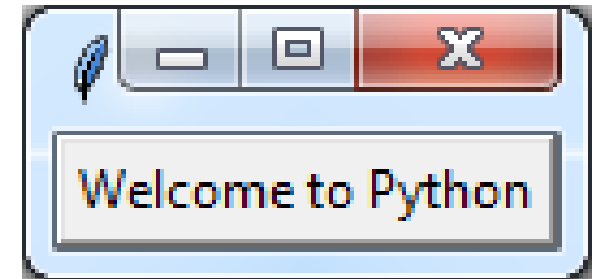
# MESSAGEBOX

```
from tkinter import *
from tkinter import messagebox
root = Tk()
root.geometry("300x200")
w = Label(root, text = 'MessageBox', font = "50")
w.pack()
messagebox.showinfo("showinfo", "Information")
messagebox.showwarning("showwarning", "Warning")
messagebox.showerror("showerror", "Error")
messagebox.askquestion("askquestion", "Are you sure?")
messagebox.askokcancel("askokcancel", "Want to continue?")
messagebox.askyesno("askyesno", "Find the value?")
messagebox.askretrycancel("askretrycancel", "Try again?")

root.mainloop()
```

# GUI WITH TKINTER IN PYTHON

```
import tkinter
window = tkinter.Tk()
window.title("Button GUI")
button_widget = tkinter.Button(window,text="Welcome to Python")
button_widget.pack()
tkinter.mainloop()
```



**pack():** It organizes the widgets in a block manner, and the complete available width is occupied by it. It's a conventional method to show the widgets in the window.

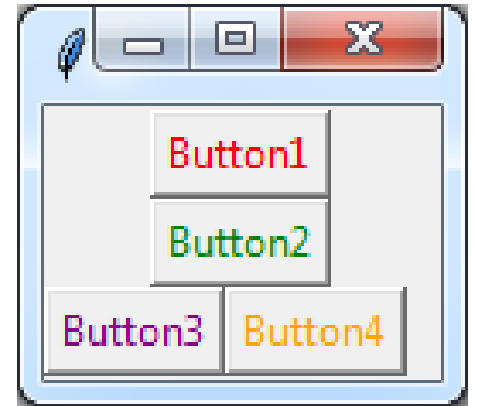


```
import tkinter
```

```
# Let's create the Tkinter window.
```

```
window = tkinter.Tk()
```

```
window.title("GUI")
```



```
# You will first create a division with the help of Frame class and align them on TOP and BOTTOM with pack() method.
```

```
top_frame = tkinter.Frame(window).pack()
```

```
bottom_frame = tkinter.Frame(window).pack(side = "bottom")
```

```
# Once the frames are created then you are all set to add widgets in both the frames.
```

```
btn1 = tkinter.Button(top_frame, text = "Button1", fg = "red").pack() #'fg or foreground' is for coloring the contents (buttons)
```

```
btn2 = tkinter.Button(top_frame, text = "Button2", fg = "green").pack()
```

```
btn3 = tkinter.Button(bottom_frame, text = "Button3", fg = "purple").pack(side = "left") #'side' is used to left or right align the widgets
```

```
btn4 = tkinter.Button(bottom_frame, text = "Button4", fg = "orange").pack(side = "left")
```

# MESSAGEBOX.ASKQUESTION

```
import tkinter
import tkinter.messagebox
```

```
# Let's create the Tkinter window
```

```
window = tkinter.Tk()
window.title("GUI")
```

```
# Let's also create a question for the user and based upon the response [Yes or No Question] display a message.
```

```
response = tkinter.messagebox.askquestion("Tricky Question", "Do you love Deep Learning?")
```

```
# A basic 'if/else' block where if user clicks on 'Yes' then it returns 'yes' else it returns 'no'.
```

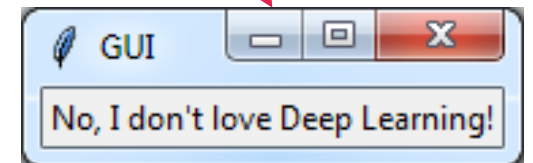
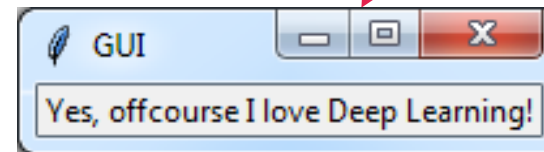
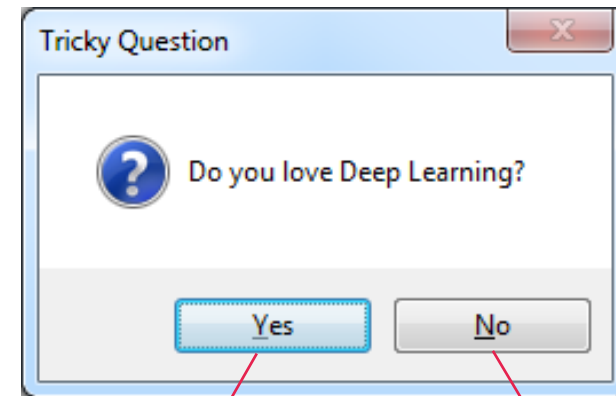
```
if response == 'yes':
```

```
    tkinter.Label(window, text = "Yes, offcourse I love Deep Learning!").pack()
```

```
else:
```

```
    tkinter.Label(window, text = "No, I don't love Deep Learning!").pack()
```

```
window.mainloop()
```



```
import matplotlib.pyplot as plt  
plt.plot([1,2,3,4], [1,4,9,16], 'ro')  
plt.axis([0, 6, 0, 20])  
plt.show()
```

