python™

For Scientific Computing

Introduction

Who is this for?

This a two day workshop introducing Python as a scientific computing language.

It is intended for those who are completely new to programming in Python.

As well as an informal lecture style delivery we will also be doing exercises.

This is based on popular Python based university courses at Cambridge and UCL.

In praise of Python

- Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.
- An excellent way to see how Python code works is to run the Python interpreter and type code right into it.

Scientific Computing

- Science has traditionally consisted of two major disciplines, theory and experimentation.

- In the last several decades a third important and exciting component has emerged, i.e. *scientific computing*.

- Scientific computing acts as an intersection of the former two areas of science. It is often closely related to theory, but it also has many characteristics in common with experimental work.

- It is therefore often viewed as a third branch of science.

The Need for Scientific Computing

- In most areas of science, computation is an invaluable complement to both experiments and theory.

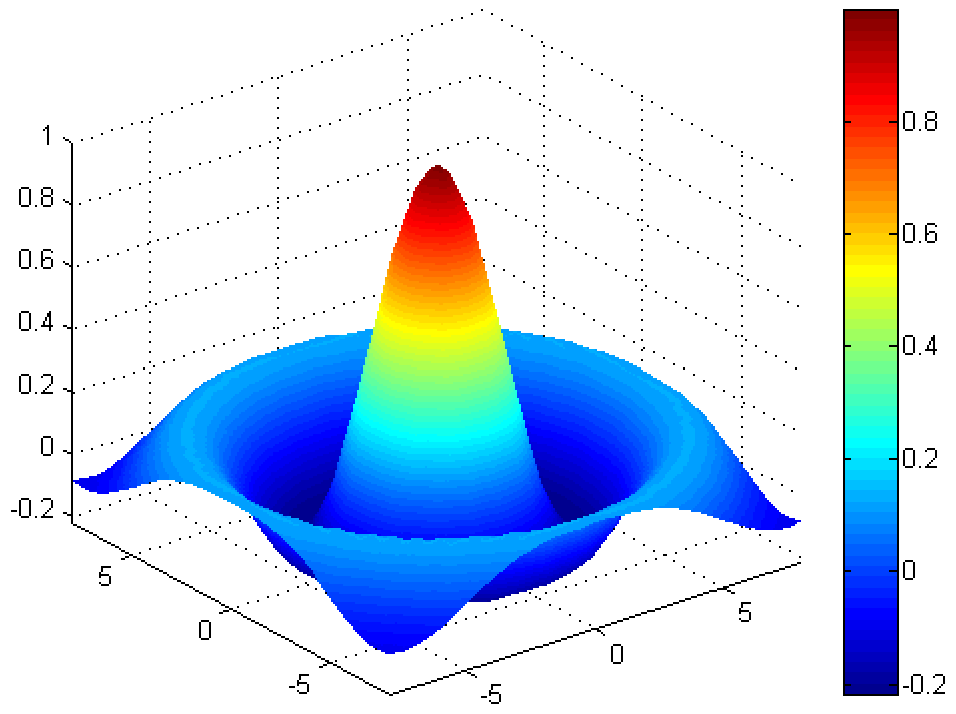
- Vast majority of both experimental and theoretical research involve some numerical calculations, simulations or computer modelling.

- In many studies, pure theory alone is insufficient in validating or demonstrating results. On the other hand, experimentation as a sole means of conducting an investigation may lack the scientific rigour necessary to hold up to scrutiny.

- Experimental work may not be possible or may prove too costly.

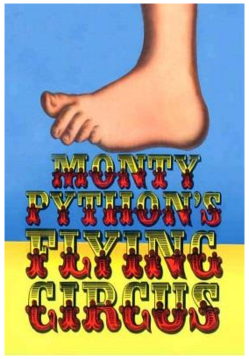
Mathematical Modelling

Most problems based on real life are often too complex and cannot be solved using analytical techniques alone – without computational methods and scientific computing techniques we would be greatly limited to the types of modelling possible.



Why Python?

Python is a general-purpose programming language initially developed by Guido van Rossum in the 1990s.
The name has nothing to do with the reptile!



Features:

- (very) Simple to pick up language – easy to maintain
- **open-source** and free language
- **multi-platform** - available on Windows, Mac OS, Linux and almost all operating systems (Android also)
- used by Google, YouTube, Instagram, NASA, CERN, Disney, . . .

The popularity of Python

Who uses Python?

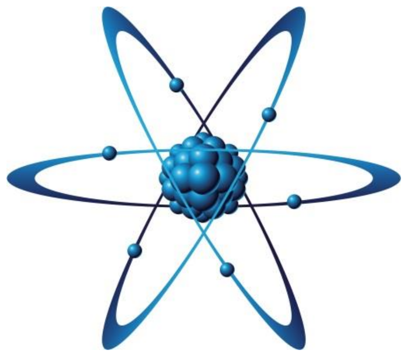
Web applications and internet

On-line games

Finance

Embedded systems

Science



Instrument control

Where does Python programing fit into your life?

- Super small so shows up on embedded devices
- Several libraries for building great web apps
- Has a strong position in scientific computing. Heavily used in science (CERN and NASA) with dedicated libraries to specific areas
  - NumPy and SciPy – general purposes
  - EarthPy – earth sciences
  - AstroPy – Astronomy
  - Pygame – writing video games; supports art, music, sound, video projects, mouse and keyboard interaction
- Python popular at Disney and Lucas Film
- Large community of users, easy to find help and documentation

Interpreted versus Compiled 1

Programs are indirectly executed by an interpreter program which reads the source code and translates it while in motion, into a series of computations and system calls. The source has to be re-interpreted (and the interpreter present) each time the code is executed.

- Slower than compiled, with limited access to the underlying operating system and hardware
- Easier to program
- Less strict on coding errors

Interpreted versus **Compiled** 2

Most conventional kind of language is compiled. Programs are converted into machine code by a compiler and then directly executed. This executable file can be run without the need to refer back to the source code.

- Give excellent performance with limited access to the underlying operating system and hardware
- Can be difficult to program in
- Most of the software we use is delivered as compiled binaries, from software which the use doesn’t see

Installation of Canopy Express

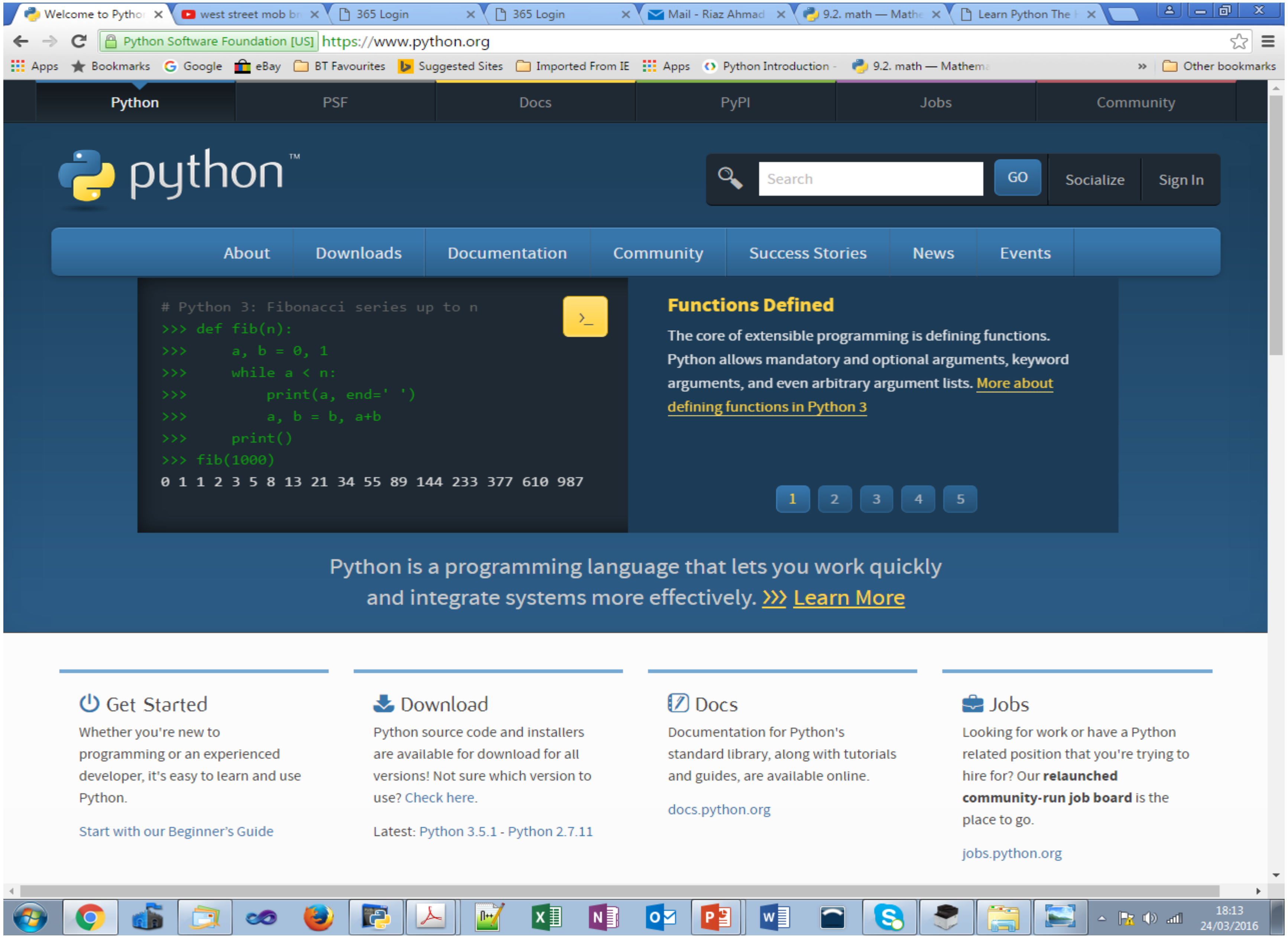
From the link

https://store.enthought.com/downloads

and download the version adapted to your system of Canopy Express.

How to know if your PC/laptop is a 32 or 64-bit system
Linux **uname -a** and if you see x64 then 64-bit else 32-bit
Mac **uname -a** also
Windows Click Start, right-click **My** Computer, and then click **Properties** if you see 64-bit it's a 64-bit system

https://www.python.org/



```
# Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Functions Defined

The core of extensible programming is defining functions. Python allows mandatory and optional arguments, keyword arguments, and even arbitrary argument lists. More about defining functions in Python 3

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Get Started

Whether you're new to programming or an experienced developer, it's easy to learn and use Python.

Start with our Beginner's Guide

Download

Python source code and installers are available for download for all versions! Not sure which version to use? Check here.

Latest: Python 3.5.1 - Python 2.7.11

Docs

Documentation for Python's standard library, along with tutorials and guides, are available online.

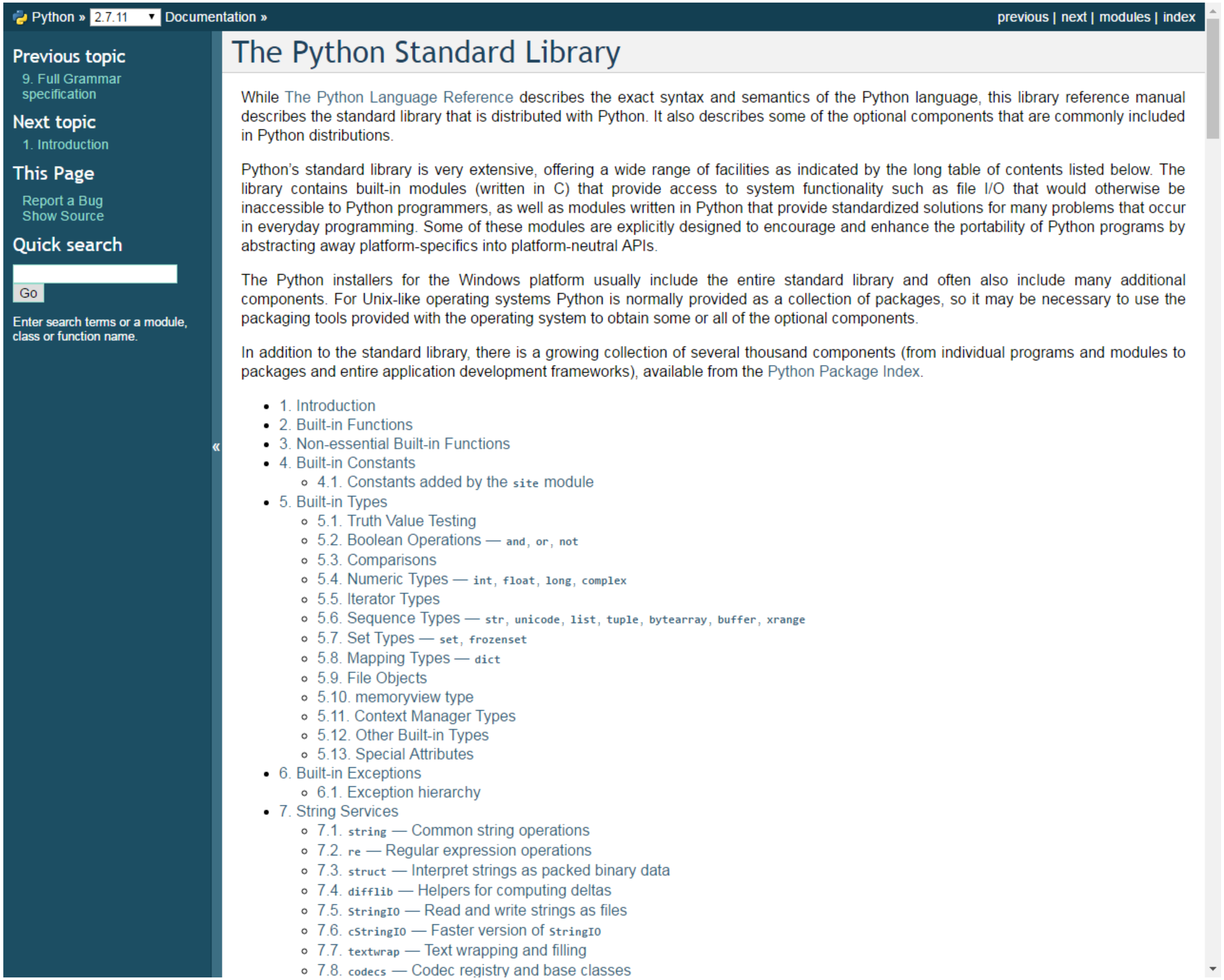
docs.python.org

Jobs

Looking for work or have a Python related position that you're trying to hire for? Our **relaunched community-run job board** is the place to go.

jobs.python.org

https://docs.python.org/2/library/



Python » 2.7.11 Documentation »
previous | next | modules | index

Enter search terms or a module, class or function name.

The Python Standard Library

While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

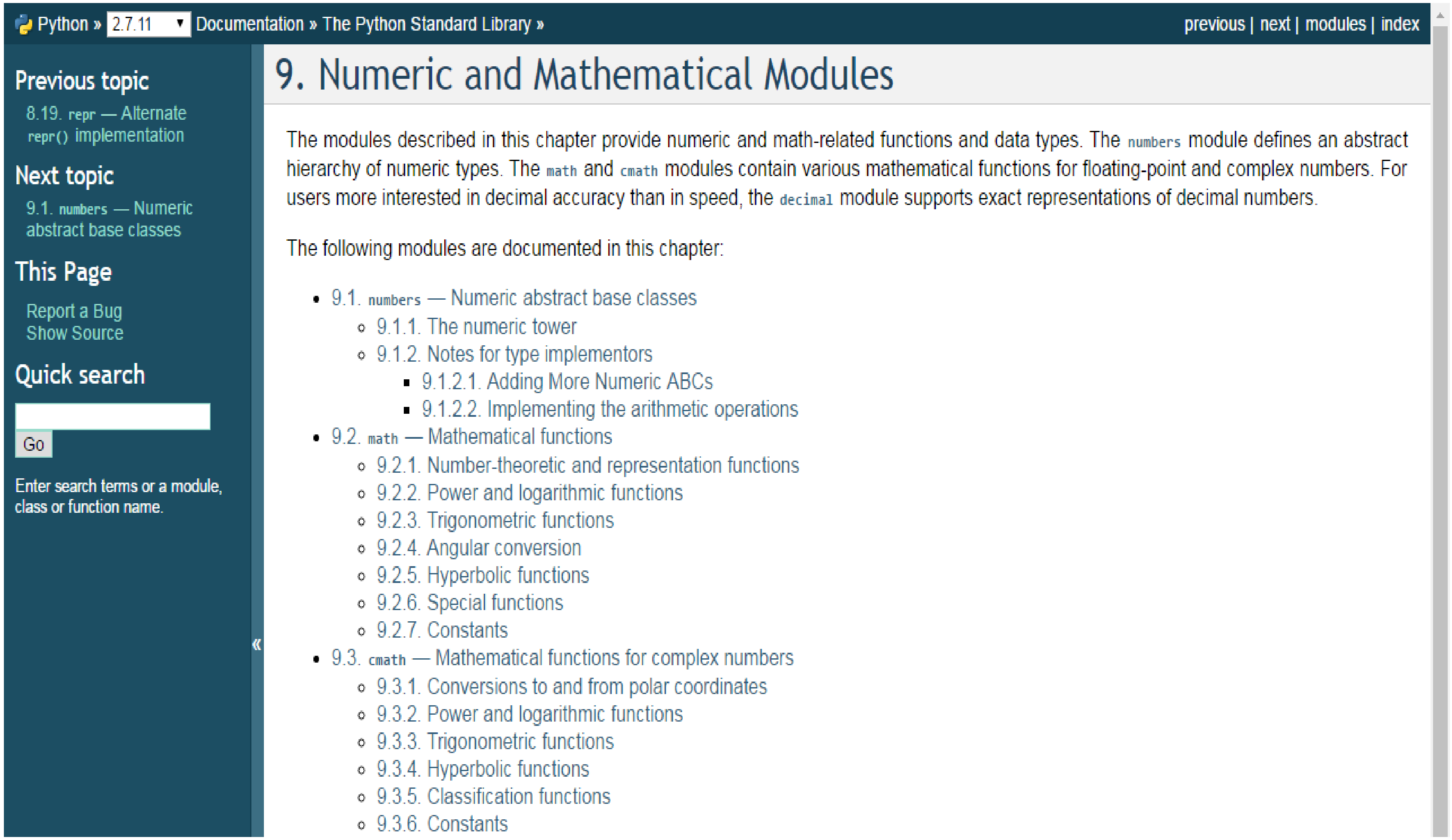
Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

Welcome to your new best friend!

https://docs.python.org/2/library/numeric.html



Python » 2.7.11 Documentation » The Python Standard Library »
previous | next | modules | index
Previous topic
8.19. repr — Alternate repr() implementation
Next topic
9.1. numbers — Numeric abstract base classes
This Page
Report a Bug
Show Source
Quick search
Go
Enter search terms or a module, class or function name.

9. Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The `numbers` module defines an abstract hierarchy of numeric types. The `math` and `cmath` modules contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the `decimal` module supports exact representations of decimal numbers.

The following modules are documented in this chapter:

What sort of language is Python?

Compiled			**Interpreted**
Explicitly compiled to machine code	Explicitly compiled to byte code	Implicitly compiled to byte code	Purely interpreted
C, C++, Fortran	Java, C#	**Python**	Shell, Perl

Distributions of Python

Python is a language, then there exist different **distributions** of Python (Python official, Anaconda Python, Enthought Canopy, pythonxy . . .).

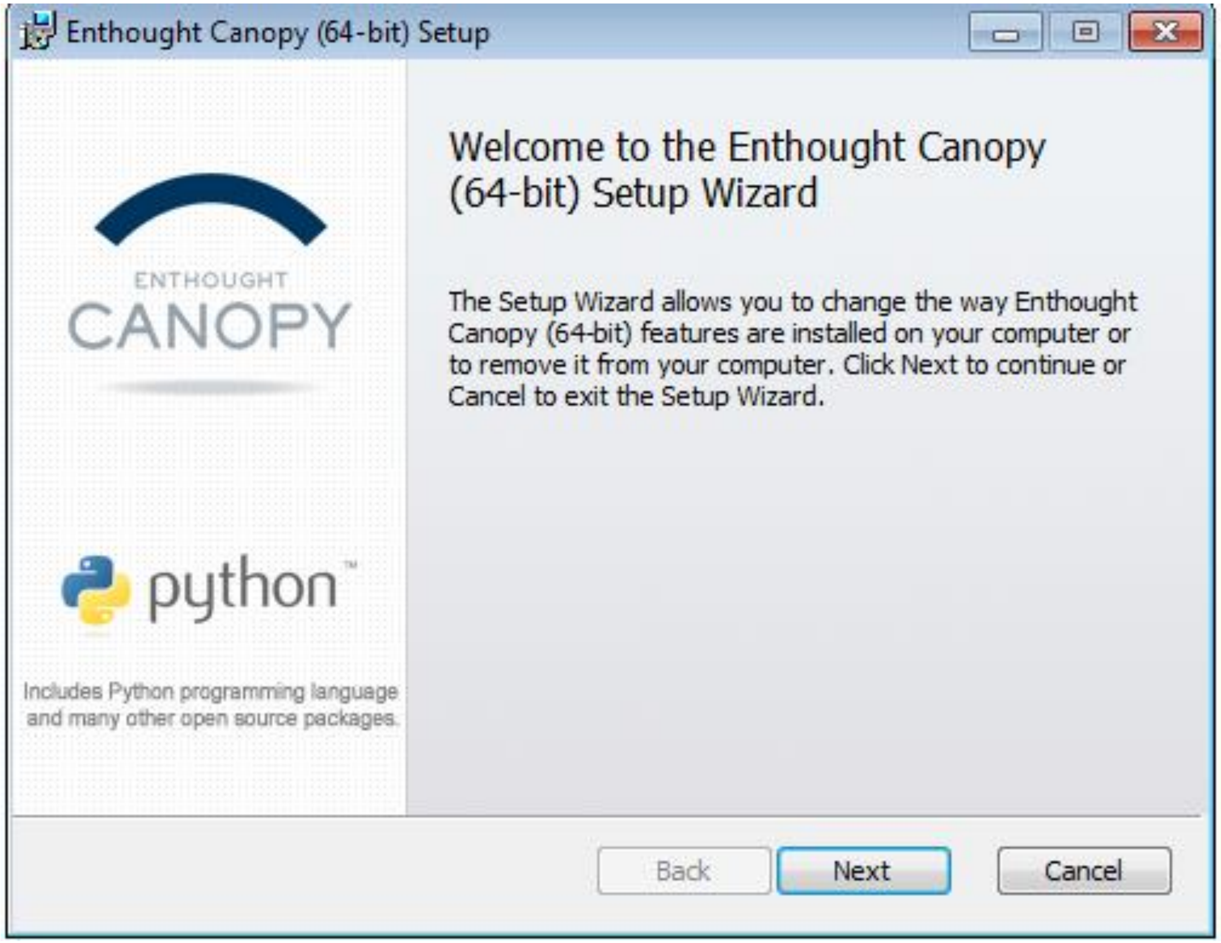
Each distribution provides specific tools (Editors, **packages** pre-installed, support for a given platform . . .).

On Linux and Mac OS, Pythons comes pre-installed with the operating system. However, many useful packages (e.g. SciPy) must be installed by hand. For its easy interface we will use **Enthought Canopy**.

Python 2 or 3

- Two Python versions are in current use: Python 2 and Python 3.
- Python 3 is not backward compatible with Python 2.
- The largest community is still using Python 2 since there is still several packages incompatible with Python 3.
- That's why we will use Python 2 (more precisely Python 2.7.6).

Installation on Windows

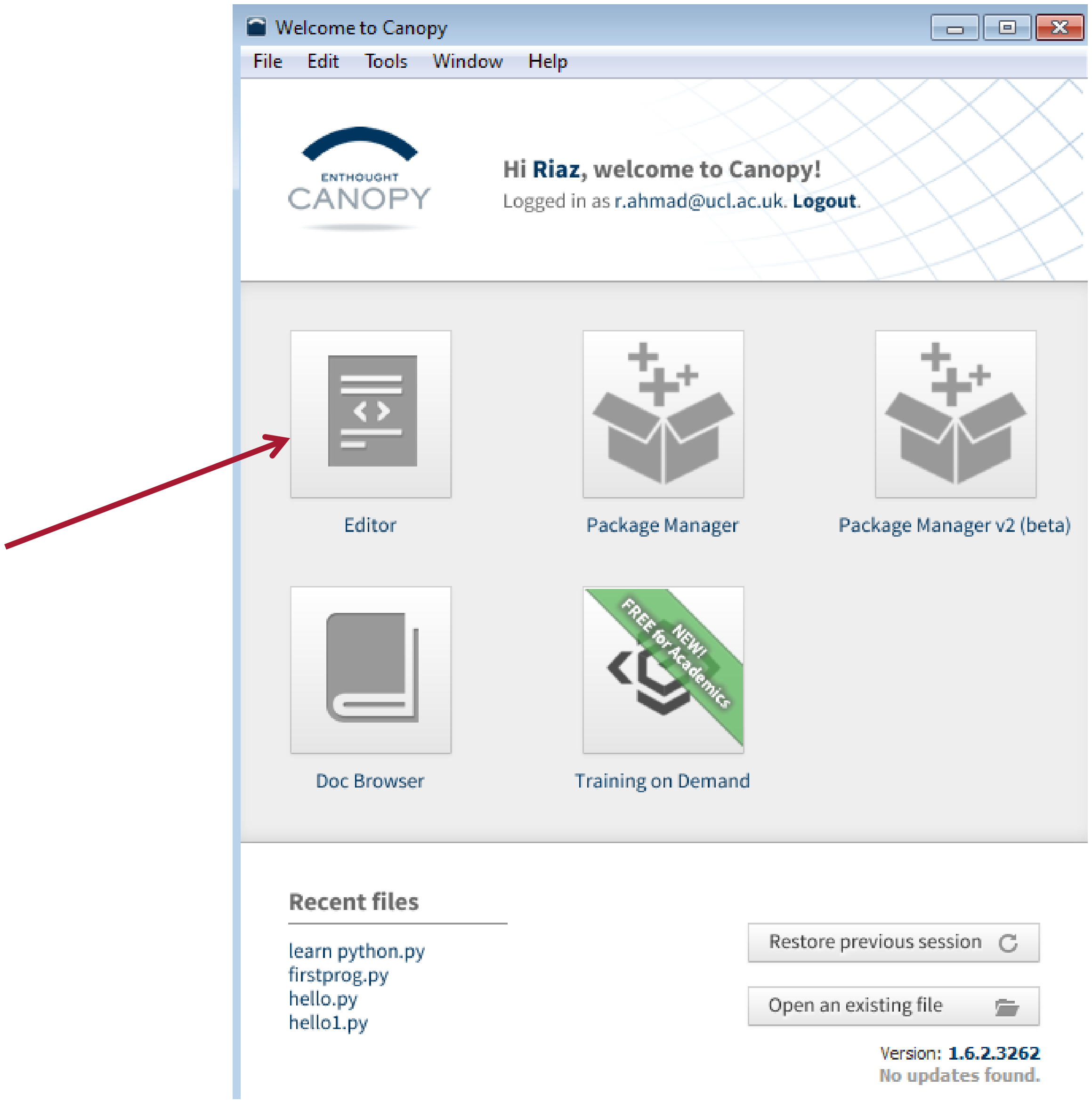

Enthought Canopy (64-bit) Setup

Welcome to the Enthought Canopy (64-bit) Setup Wizard

The Setup Wizard allows you to change the way Enthought Canopy (64-bit) features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Includes Python programming language and many other open source packages.

Getting started with Canopy



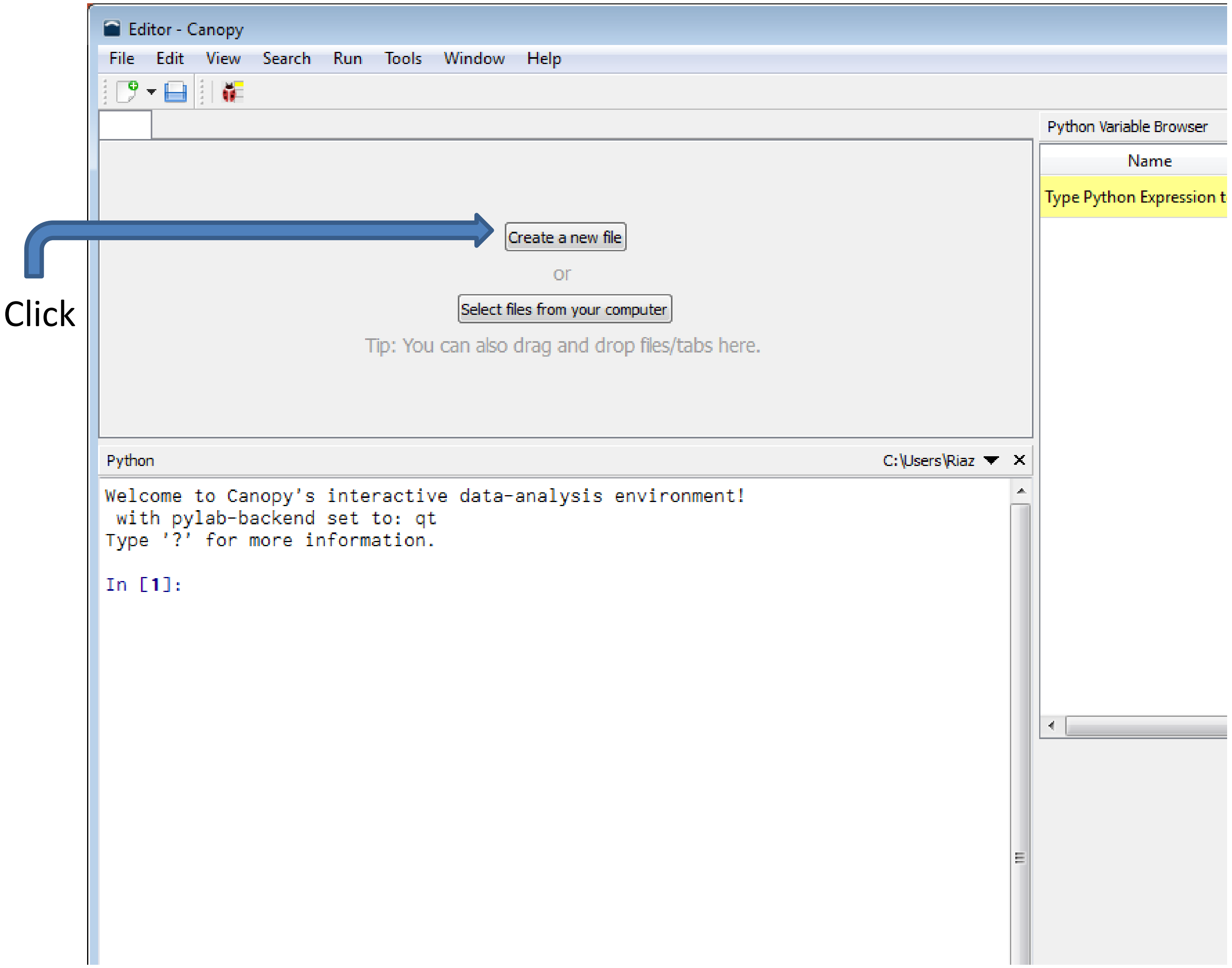
Hi Riaz, welcome to Canopy!
Logged in as r.ahmad@ucl.ac.uk. Logout.

Recent files

learn python.py
firstprog.py
hello.py
hello1.py

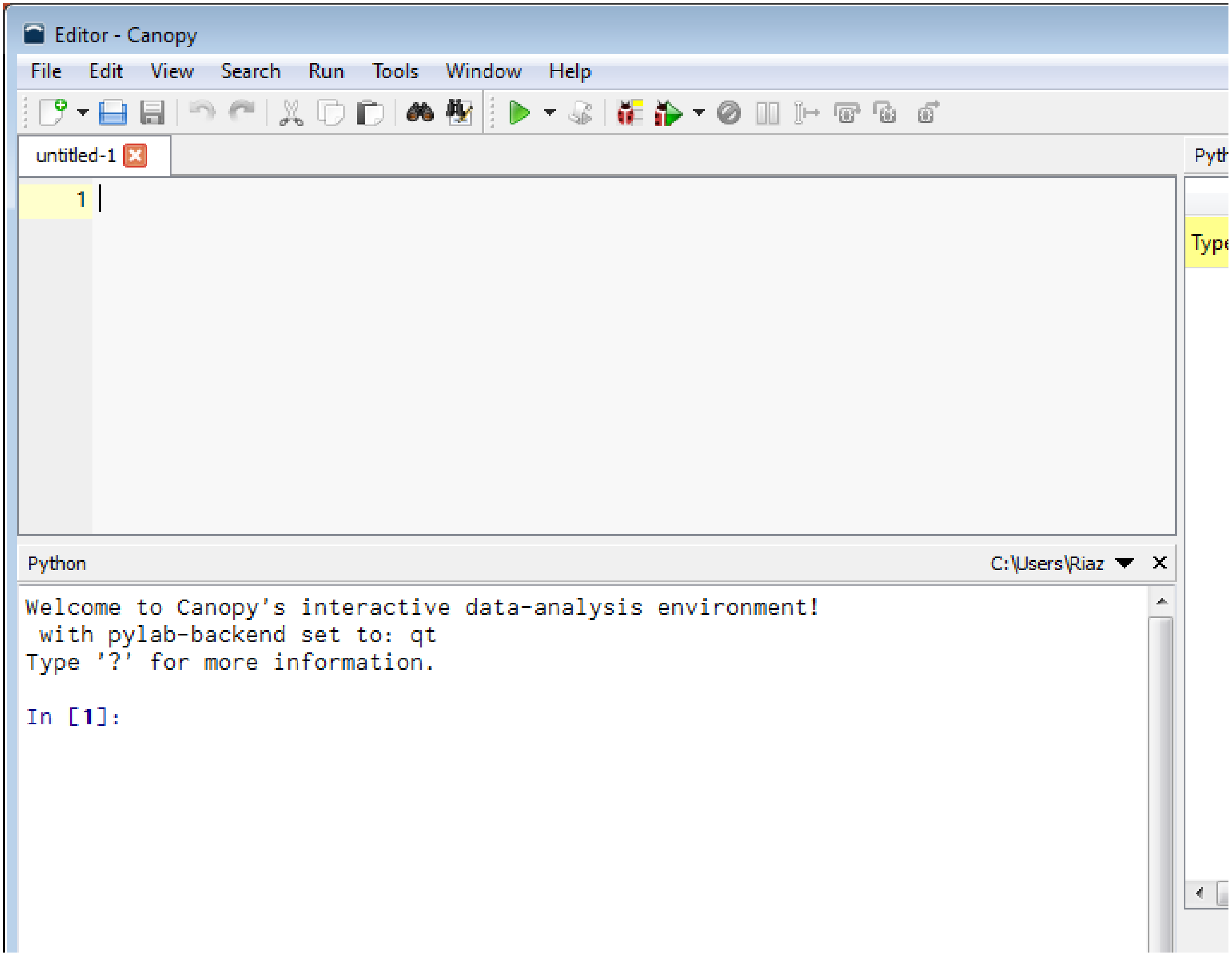
Version: 1.6.2.3262
No updates found.

Editing Environment


Editor - Canopy
File Edit View Search Run Tools Window Help
Python Variable Browser
Name
Type Python Expression t
Create a new file
or
Select files from your computer
Tip: You can also drag and drop files/tabs here.
Python
C:\Users\Riaz
Click

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]:
```

Ready to start programming!



```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]:
```

Syntactic sugar

syntactic sugar is **syntax** within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

https://en.wikipedia.org/wiki/Syntactic_sugar

Python script

Python script is a set of instructions, written in the Python language, that run in order and carry out a series of commands.

Importing Modules

When a Python program starts it only has access to a basic functions and classes.

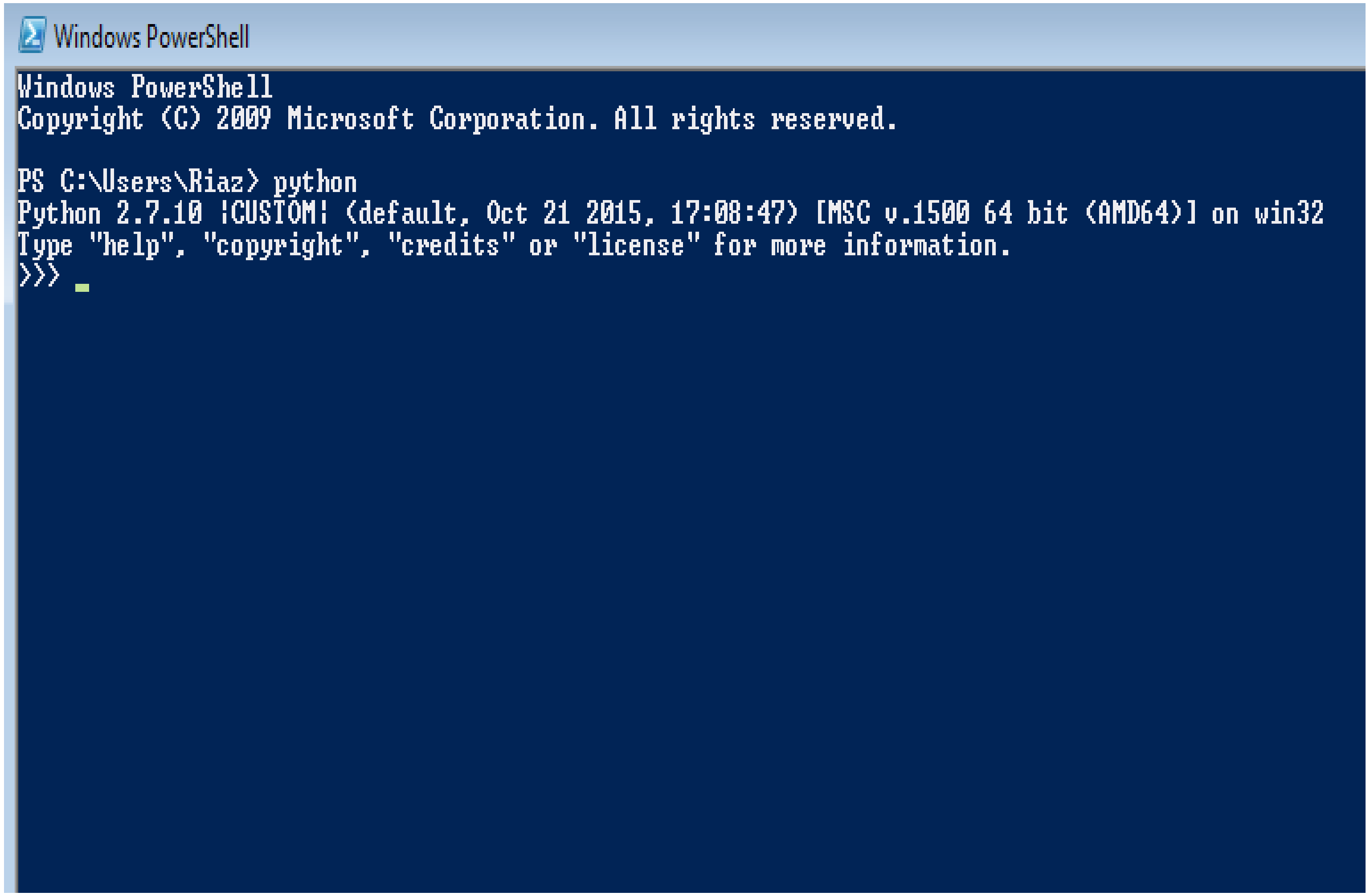
("int", "dict", "len", "sum", "range", ...)

"Modules" contain additional functionality.

Use "import" to tell Python to load a module.

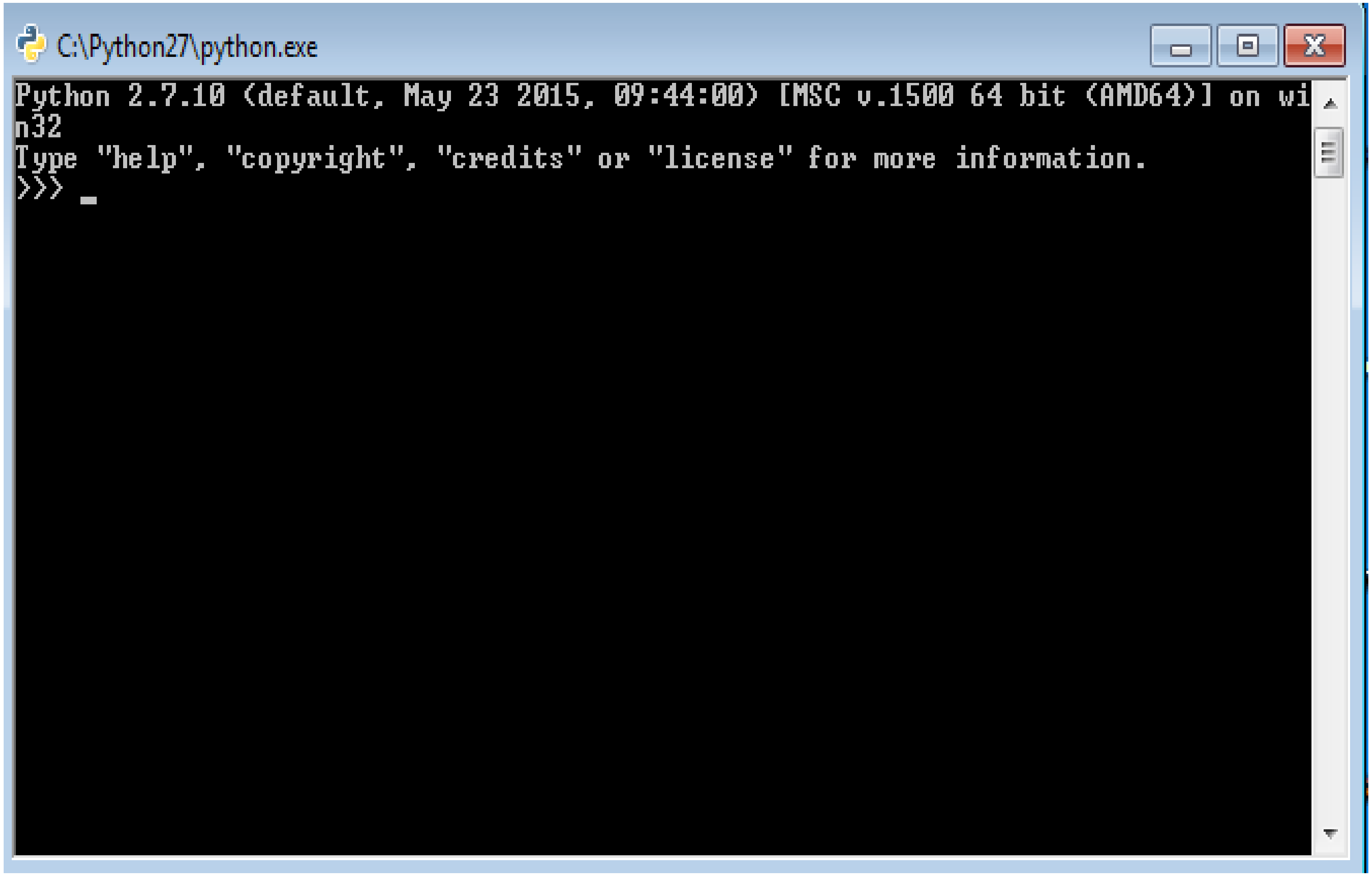
```
>>> import math
>>> import nltk
```

Type Powershell in the start menu



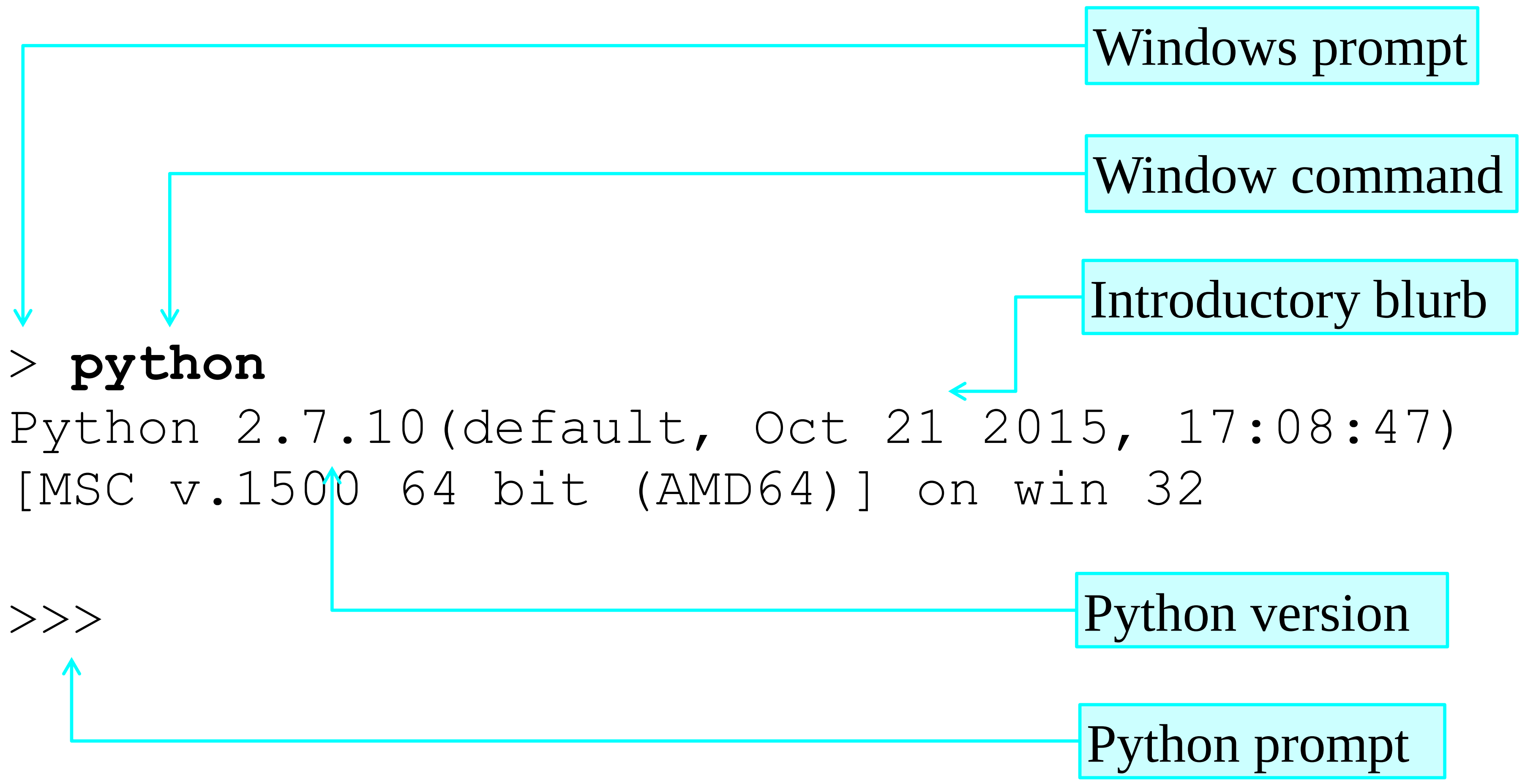
```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Riaz> python
Python 2.7.10 |CUSTOM| (default, Oct 21 2015, 17:08:47) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Command Line



```
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on wi
n32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Running Python



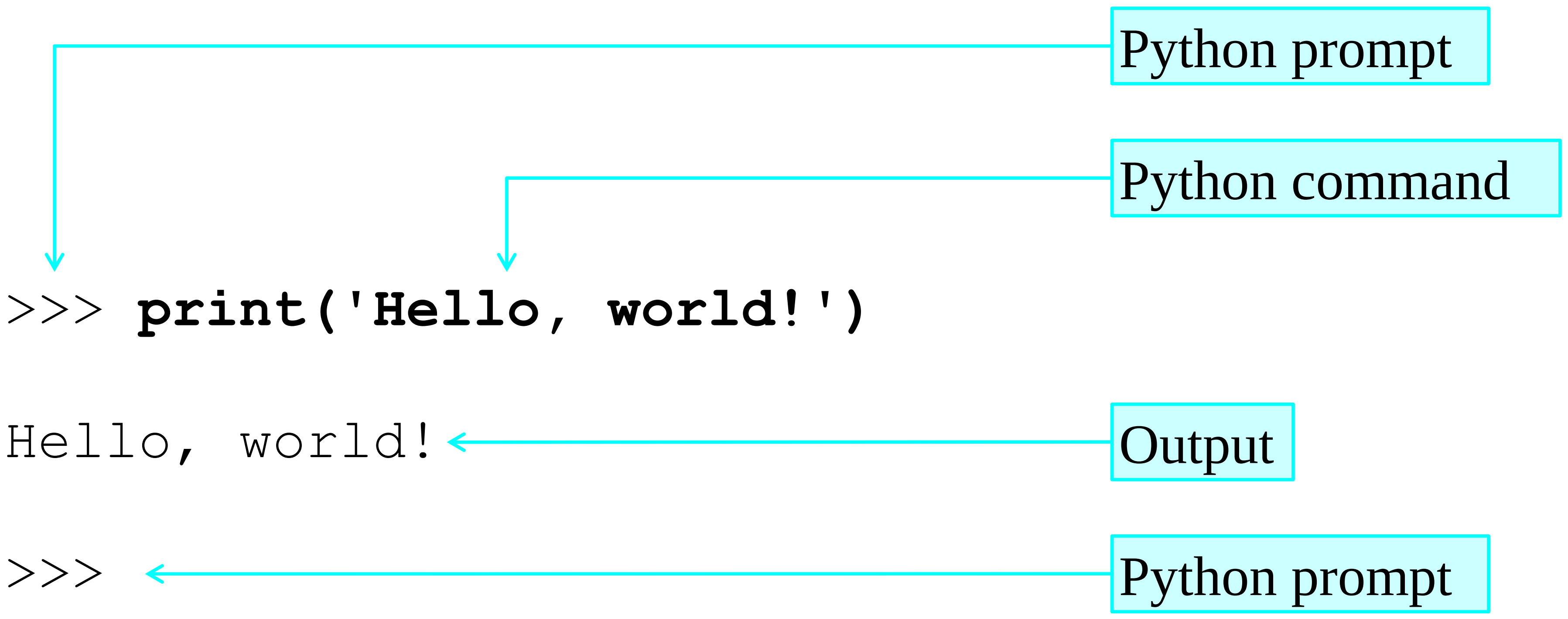
```
> python
Python 2.7.10(default, Oct 21 2015, 17:08:47)
[MSC v.1500 64 bit (AMD64)] on win 32

>>>
```

Quitting Python

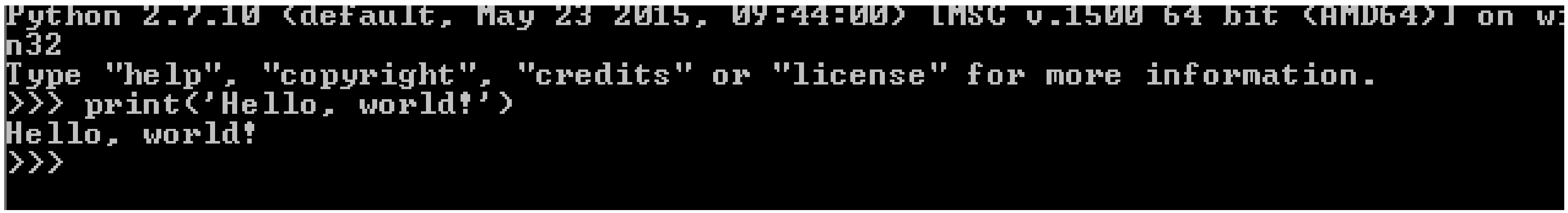
```
>>> exit()

>>> quit()

>>> Ctrl + D
```

Any one of these

Welcome to Python – first program



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

Hello, world!

>>>
```



```
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on wi
n32
Type "help", "copyright", "credits" or "license" for more information.
>>> print(’Hello, world!’)
Hello, world!
>>>
```

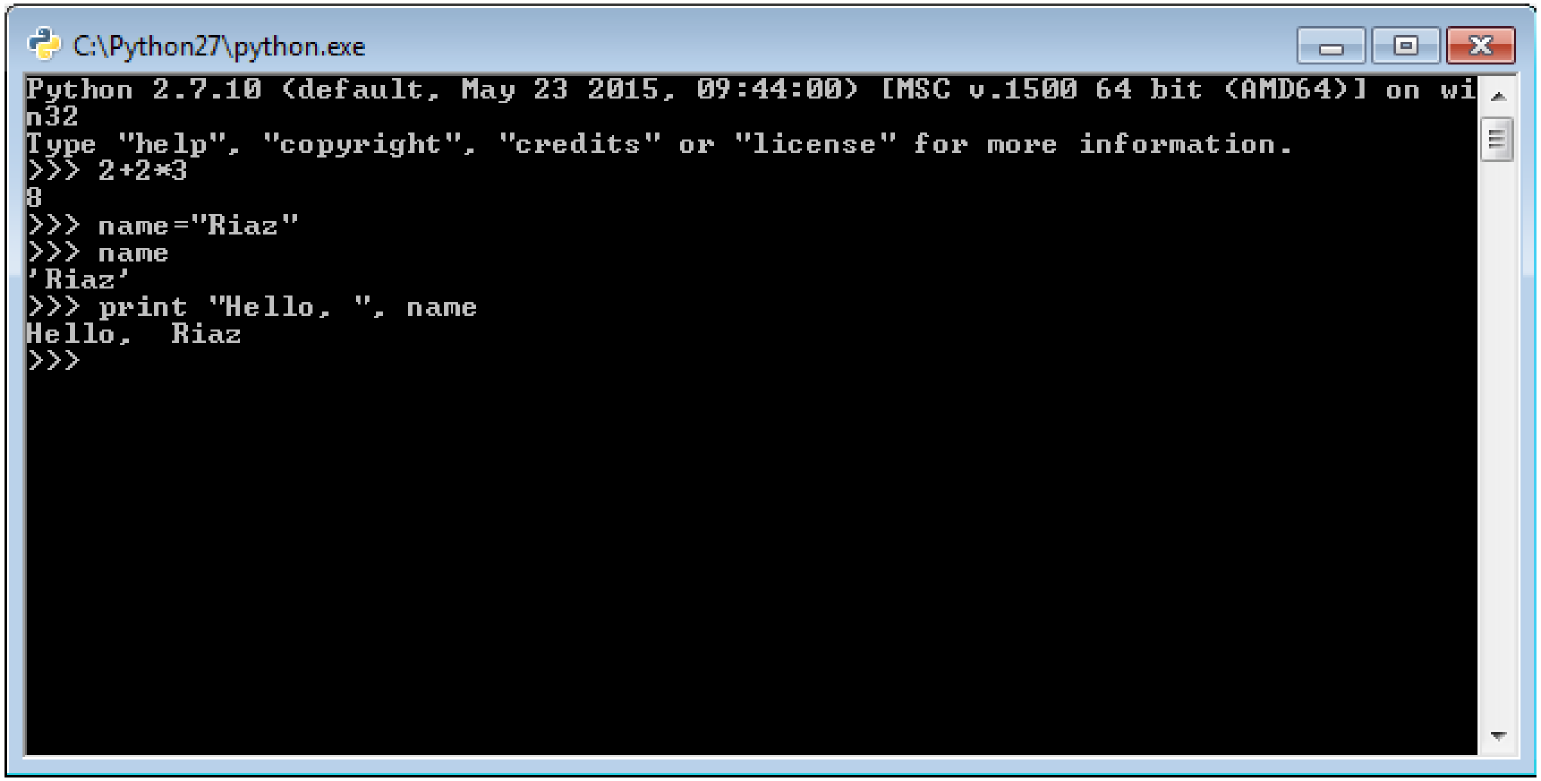
```
>>> print 'Hello, world!'
Hello, world!
>>>
```

We note that the same outcome is obtained without the use of parentheses.
Brackets are used for consistency with version 3

A string is a sequence of characters enclosed in single or double quotes.

Python interactive shell

You can type things directly into a running Python session



```
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on wi
n32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2*3
8
>>> name="Riaz"
>>> name
'Riaz'
>>> print "Hello, ", name
Hello,  Riaz
>>>
```

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- Use a newline to end a line of code.
- No braces { } to mark blocks of code in Python… *Use consistent indentation instead*.
  - The first line with less indentation is outside of the block.

The first line with more indentation starts a nested block

- Often a colon appears at the start of a new block. (e.g. for function and class definitions.)

Assignment

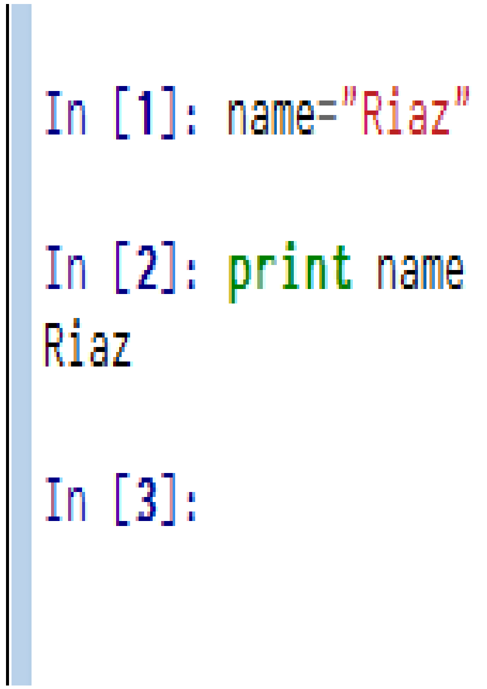
To assign a value to a **variable**, we use the operator = (not to be confused with ‘equal to’). It is evaluated from right to left

`name = “Riaz”`

can be seen as `name ← “Riaz”`

and interpreted as ‘Riaz is assigned to the variable name’.

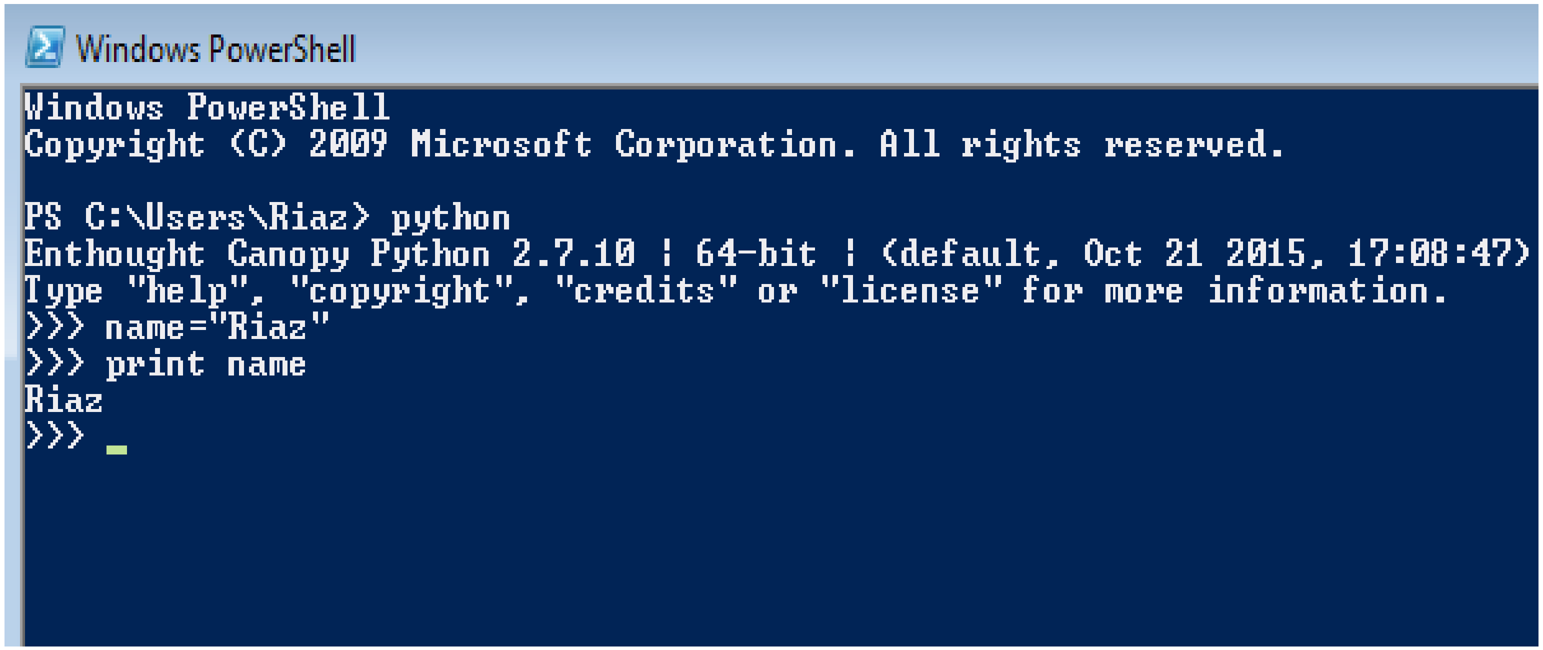
Interactive on Canopy



```
In [1]: name="Riaz"

In [2]: print name
Riaz

In [3]:
```

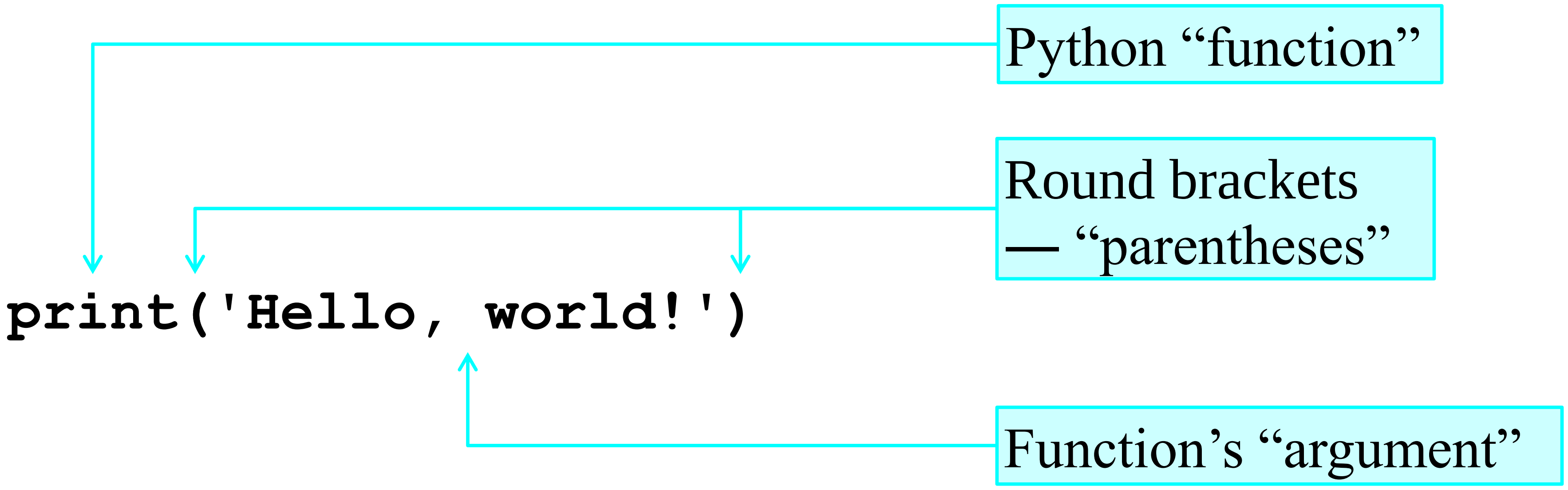
```
In [4]: print 2+2*3
8

In [5]: print "hello", name
hello Riaz
```

Interactive on PowerShell



```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Riaz> python
Enthought Canopy Python 2.7.10 | 64-bit | (default, Oct 21 2015, 17:08:47)
Type "help", "copyright", "credits" or "license" for more information.
>>> name="Riaz"
>>> print name
Riaz
>>> _
```

Python commands



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

print ≠ PRINT

“Case sensitive”

Defining strings

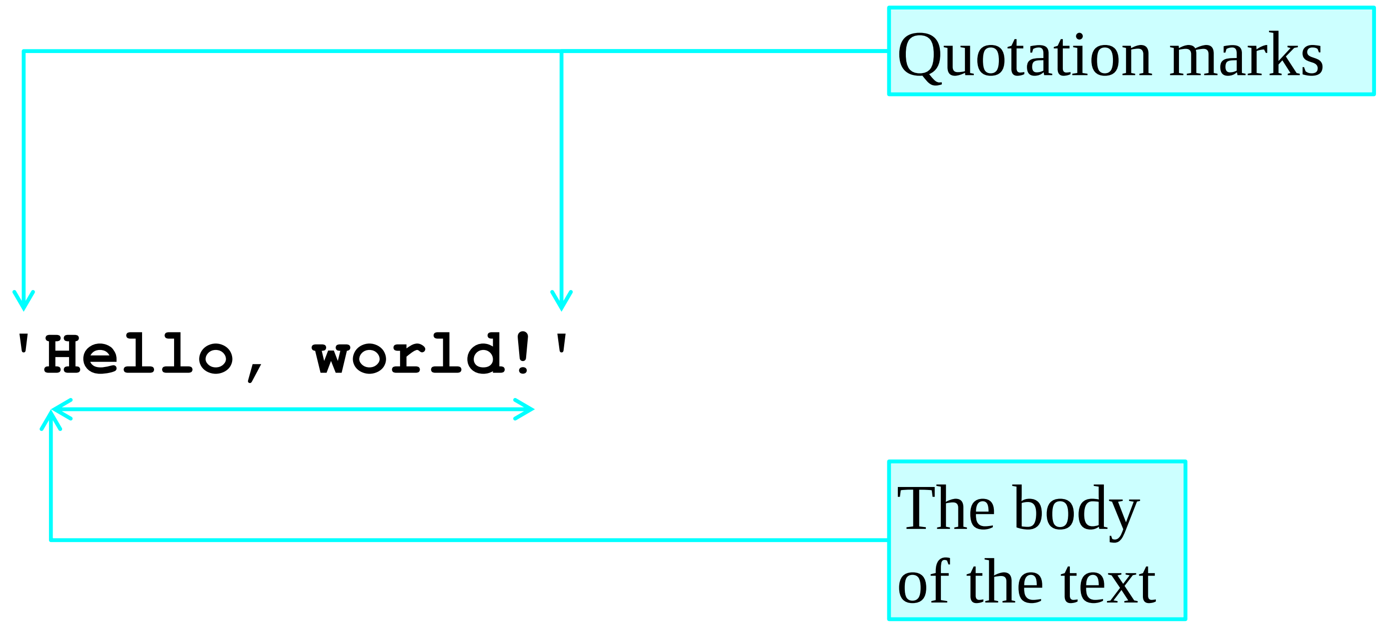
Strings can be defined using

- single quotes ' …. '

- double quotes " …. "

So strings defined using single and double quotes are identical

Python text



```
'Hello, world!'
```

The quotes are not part of the text itself.

Quotes?

$\texttt{print} \longrightarrow$ Command/statement

$\texttt{'print'} \longrightarrow$ Text/string

Manipulating Strings

- strings defined using single and double quotes are identical
- if you want to use quotes in your string, switch between single and double quotes, example:

```
print(“He said, ‘after you, Sir!’.")
```

- Anything following **#** is a **comment**
- to use \ in strings, use \\
- We can add arbitrary quotes by **escaping** them:

```
print("She said. \"He\’s coming.\".")
```

- \is **reserved** (`\n` for a newline, `\t` for a tabulation, \’ . . .)

Modules

When a Python program starts it only has access to basic functions and classes.
Most of the functionality in Python is provided by *modules*. The Python Standard Library is a collection of modules that provides cross-platform implementations of common facilities such

- I/O
- Mathematics
- String manipulation

Use “import” to tell Python to load a module e.g.

```
>>> import math
```

More on this shortly

Operations on Strings

Two strings can be joined/added together (**concatenated**) using the + **operator**:

```
In   [1]: "Hello, " + "Riaz!"

Out[1]: 'Hello, Riaz!'
```

To find out the length of a string, use **the function** len():

```
In   [2]: len(“supercalifragilisticexpialidocious”) # from Mary Poppins

Out [2]: 34
```

We can join several copies of a string with the **operator** *

```
In   [3]: 3 * "AbC"

Out [3]: 'AbCAbCAbC'
```

Slicing

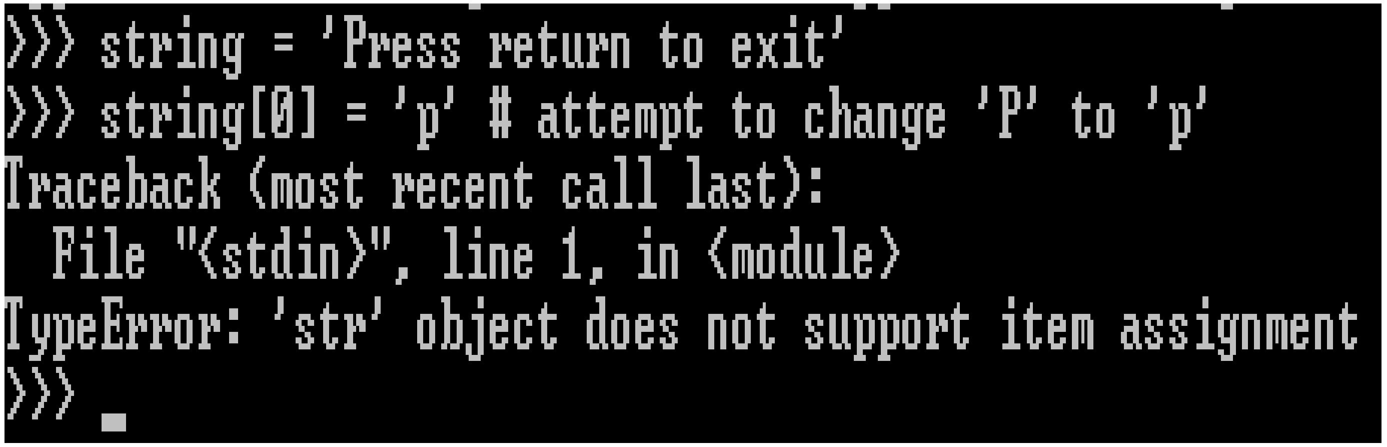
Slicing is used to extract a portion of the string. Here is an example:

```
>>> string = 'Press return to exit'

>>> print string[0:12]

Press return␣


string[12]
```

Immutability

A string is an *immutable* object. Its individual characters cannot be modified with an assignment statement and it has a fixed length. Any attempt to be violate this property will result in `TypeError`

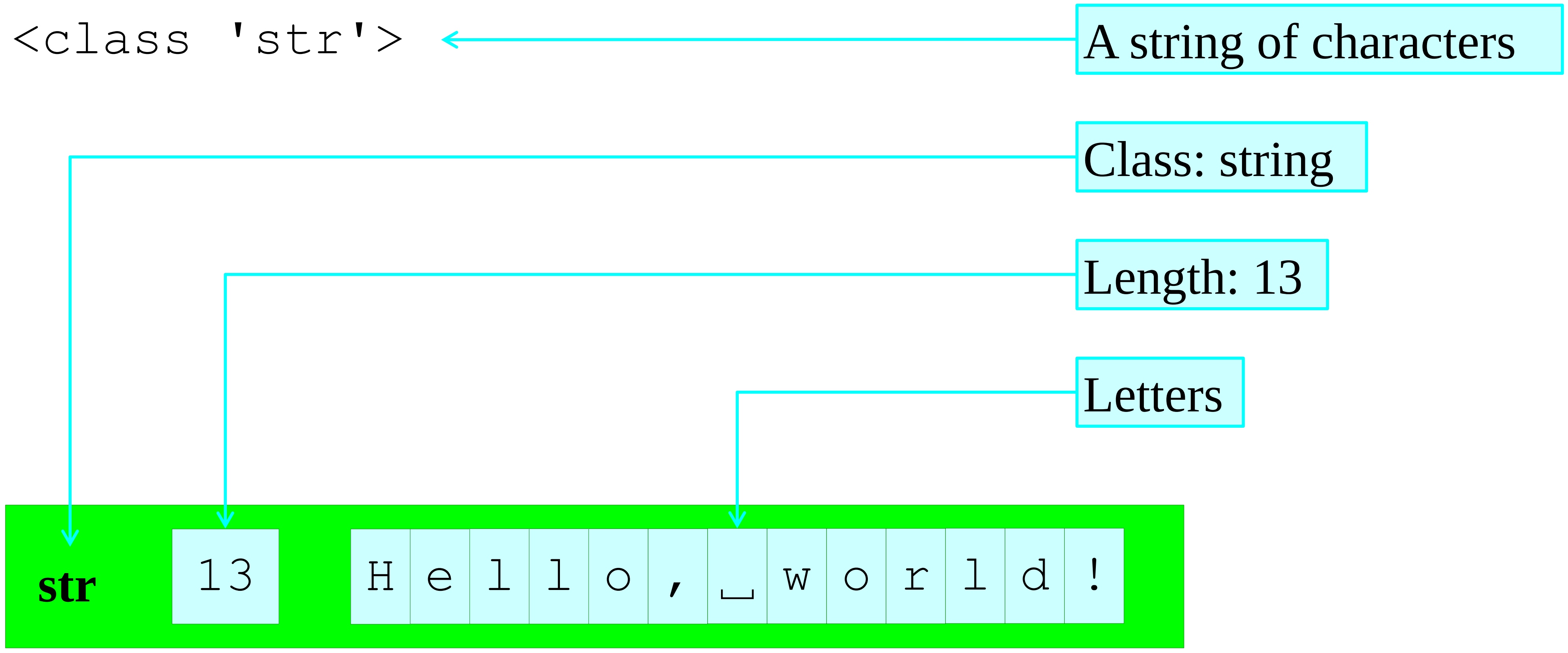
Consider the code below:



```
>>> string = 'Press return to exit'
>>> string[0] = 'p' # attempt to change 'P' to 'p'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> _
```

Text: a string of characters

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

<class 'str'>
```



Pure concatenation

```
>>> 'Hello,␣' + 'world!'
'Hello, world!'
```

```
>>> 'Hello,' + '␣world!'
'Hello, world!'
```

Only simple concatenation

```
>>> 'Hello,' + 'world!'
'Hello,world!'
```

No spaces added automatically.

Single & double quotes

```
>>> 'Hello, world!'
'Hello, world!'
```

Single quotes

Single quotes

```
>>> "Hello, world!"
'Hello, world!'
```

Double quotes

Single quotes

Python strings: input & output

```
'Hello, world!'

"Hello, world!"
```

Single or double quotes on input.

Create same string object.

```
'Hello, world!'
```

Single quotes on output.

Uses of single & double quotes

```
>>> print('He said "hello" to her.')

He said "hello" to her.


>>> print("He said 'hello' to her.")

He said 'hello' to her.
```

Why we need different quotes

```
>>> print('He said 'hello' to her.')

File "<stdin>", line 1
  print('He said 'hello' to her.')
                        ^
SyntaxError: invalid syntax
```

Adding arbitrary quotes

```
>>> print('He said \'hello\' to her.')
He said 'hello' to her.
```

$$\texttt{\textbackslash'} \longrightarrow \texttt{'}$$

$$\texttt{\textbackslash"} \longrightarrow \texttt{"}$$

Just an ordinary character.

“Escaping”

Putting line breaks in text

```
Hello,
world!
```

What we want

```
>>> print('Hello, ↵
world')
```

Try this

```
>>> print('Hello, ↵
File "<stdin>", line 1
  print('Hello,
               ^
SyntaxError: EOL while
scanning string literal
```

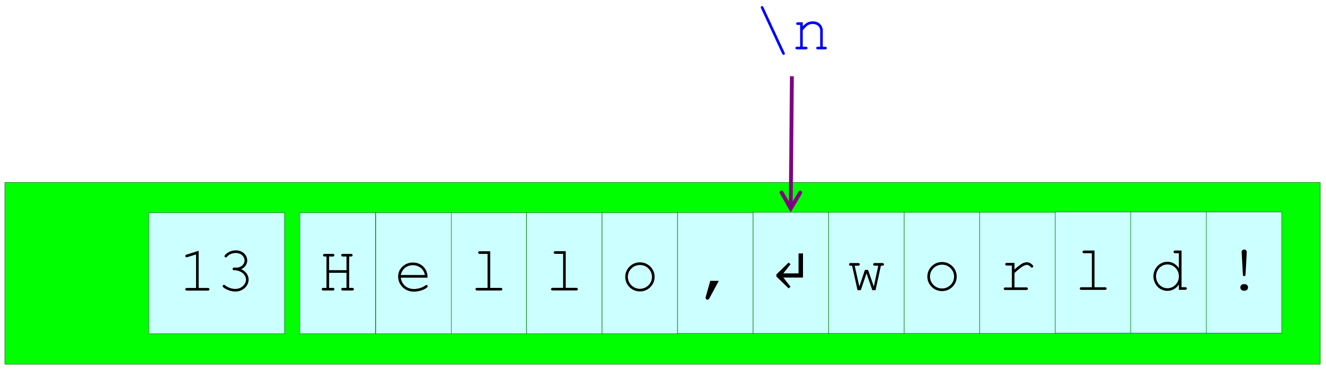
✗

“EOL”: End Of Line

Inserting “special” characters

```
>>> print('Hello,\nworld!')
Hello,
world!
```

Treated as a new line.



Converted into a *single* character.

```
>>> len('Hello,\nworld!')
13
```

`len()` function: gives the length of the object

The backslash

Special $\longrightarrow$ Ordinary

`\'` $\longrightarrow$ `'`

`\"` $\longrightarrow$ `"`

Ordinary $\longrightarrow$ Special

`\n` $\longrightarrow$ ↲

`\t` $\longrightarrow$ ⇥

`\n` Multiline strings

```
"Shoot all the blue jays you want, \nif you can
hit 'em, but remember it's a \nsin to kill a
mockingbird. \nThat was the only time I ever
heard Atticus say \nit was a sin to do something,
\nand I asked Miss Maudie about it."
```

By default, Python assumes that the whole instruction is contained in a single line.

Special input method for long text

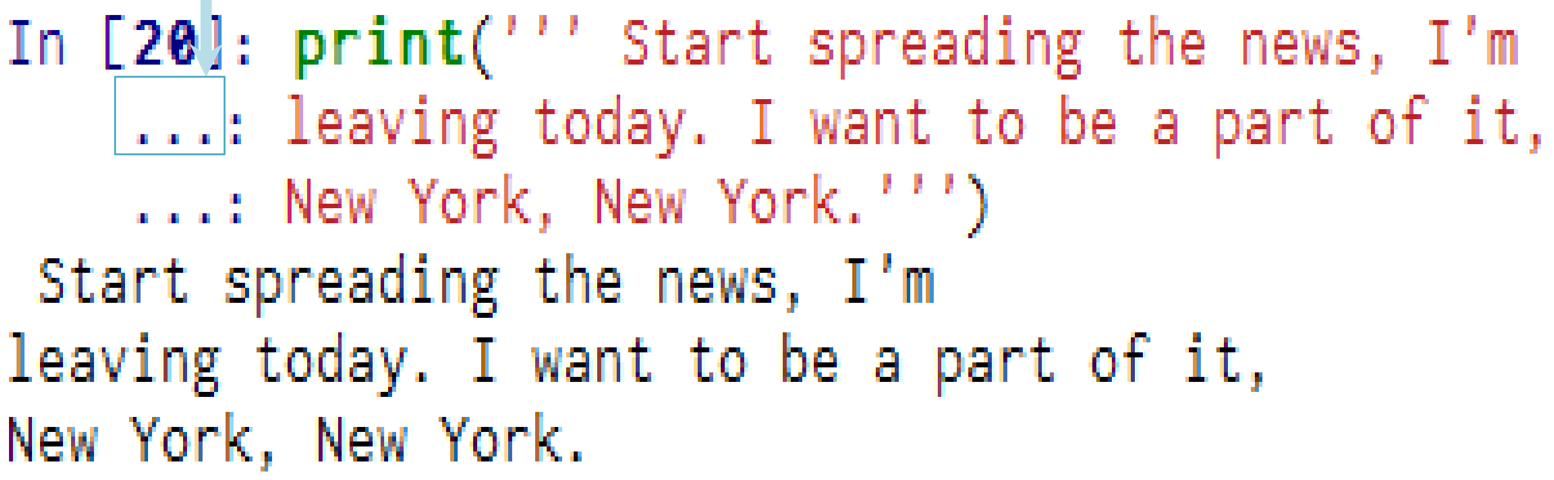
```
''' Shoot all the blue jays you
want, if you can hit 'em, but
remember it's a sin to kill a
mockingbird. That was the
only time I ever heard Atticus
it was a sin to do something,
and I asked Miss Maudie
about it.'''
```

Multiple lines

Triple quotes

Python's "secondary" prompt

Python asking for more of the same command.



```
In [20]: print(''' Start spreading the news, I'm
    ...: leaving today. I want to be a part of it,
    ...: New York, New York.''')
 Start spreading the news, I'm
leaving today. I want to be a part of it,
New York, New York.
```

It’s still just text!

```
>>> 'Hello,\nworld!'
'Hello\nworld'
```

Python uses \n to represent line breaks in strings.

```
>>> '''Hello,
... world!'''
'Hello\nworld'
```

Exactly the same!

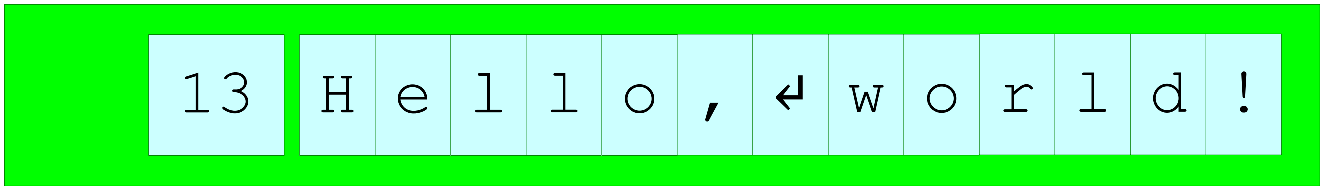
Your choice of input quotes:

Four inputs:

```
'Hello,\nworld!'                "Hello,\nworld!"
```

```
'''Hello,                       """Hello,
world!'''                       world!"""
```

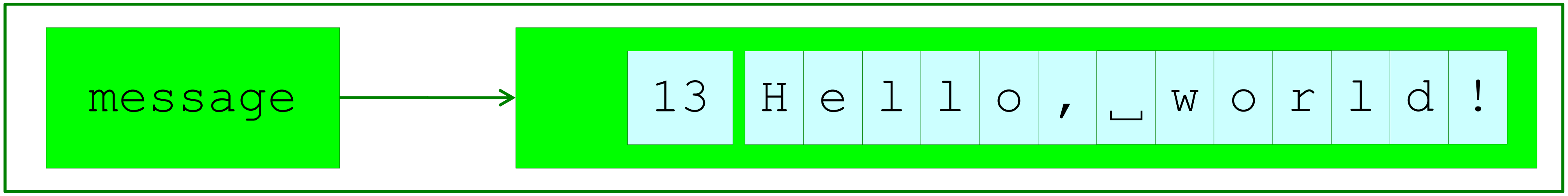
Same result:



Attaching names to values

“variables”

```
>>> message='Hello, world!'
>>> message
'Hello, world!'
>>> type(message)
<class 'str'>
```



Some more types

```
>>> type(G’day!')

<class 'str'>
```

string of characters

```
>>> type(-62)

<class 'int'>
```

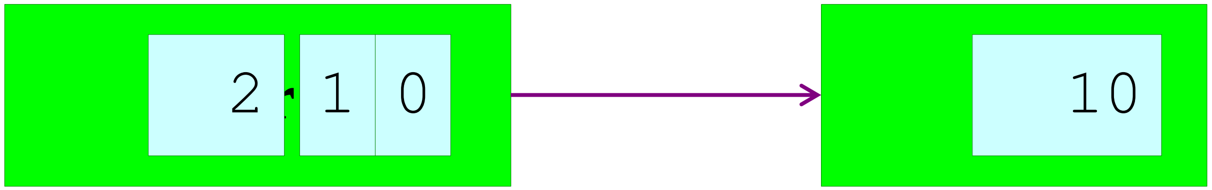
integer

```
>>> type(pi)

<class 'float'>
```

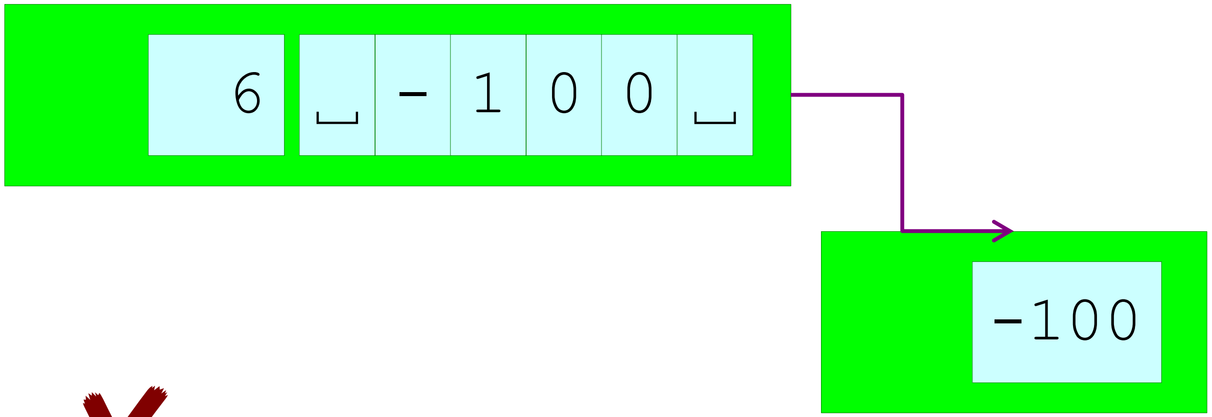
floating point number

Converting text to integers

```
>>> int('10')
10
```



```
>>> int('␣-100␣')
-100
```



```
>>> int('100-10')
```

✗

```
ValueError:
invalid literal for int() with base 10: '100-10'
```

Exercise 1.0

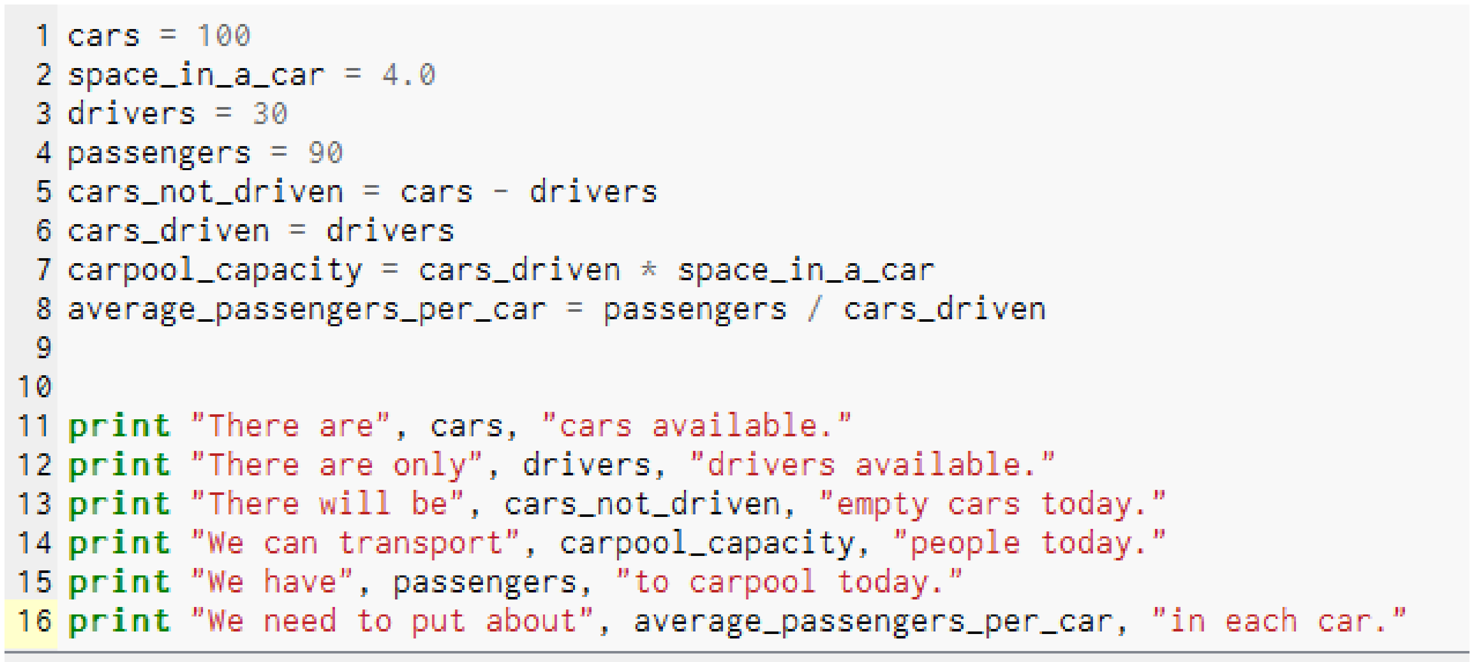
1. Write script to print the following text (with the line breaks) and then run the script.

coffee
café
caffè
Kaffee

2. Create the following output

It is a tale
Told by an idiot, full of sound and fury,
Signifying nothing.

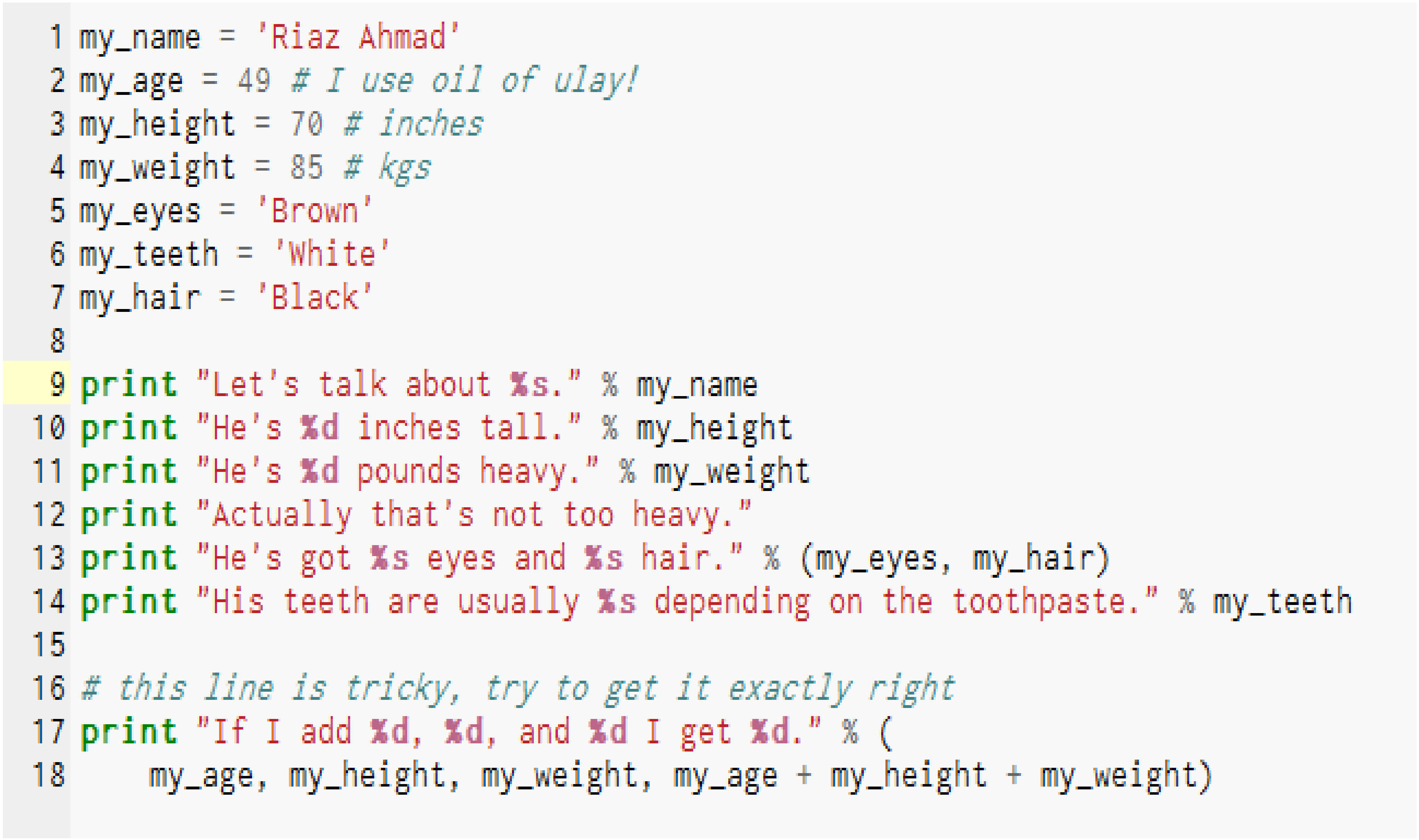
3. Create a file called car.py and write a comment next to each line explaining what it does in English



```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven


print "There are", cars, "cars available."
print "There are only", drivers, "drivers available."
print "There will be", cars_not_driven, "empty cars today."
print "We can transport", carpool_capacity, "people today."
print "We have", passengers, "to carpool today."
print "We need to put about", average_passengers_per_car, "in each car."
```

Exercise 1.1

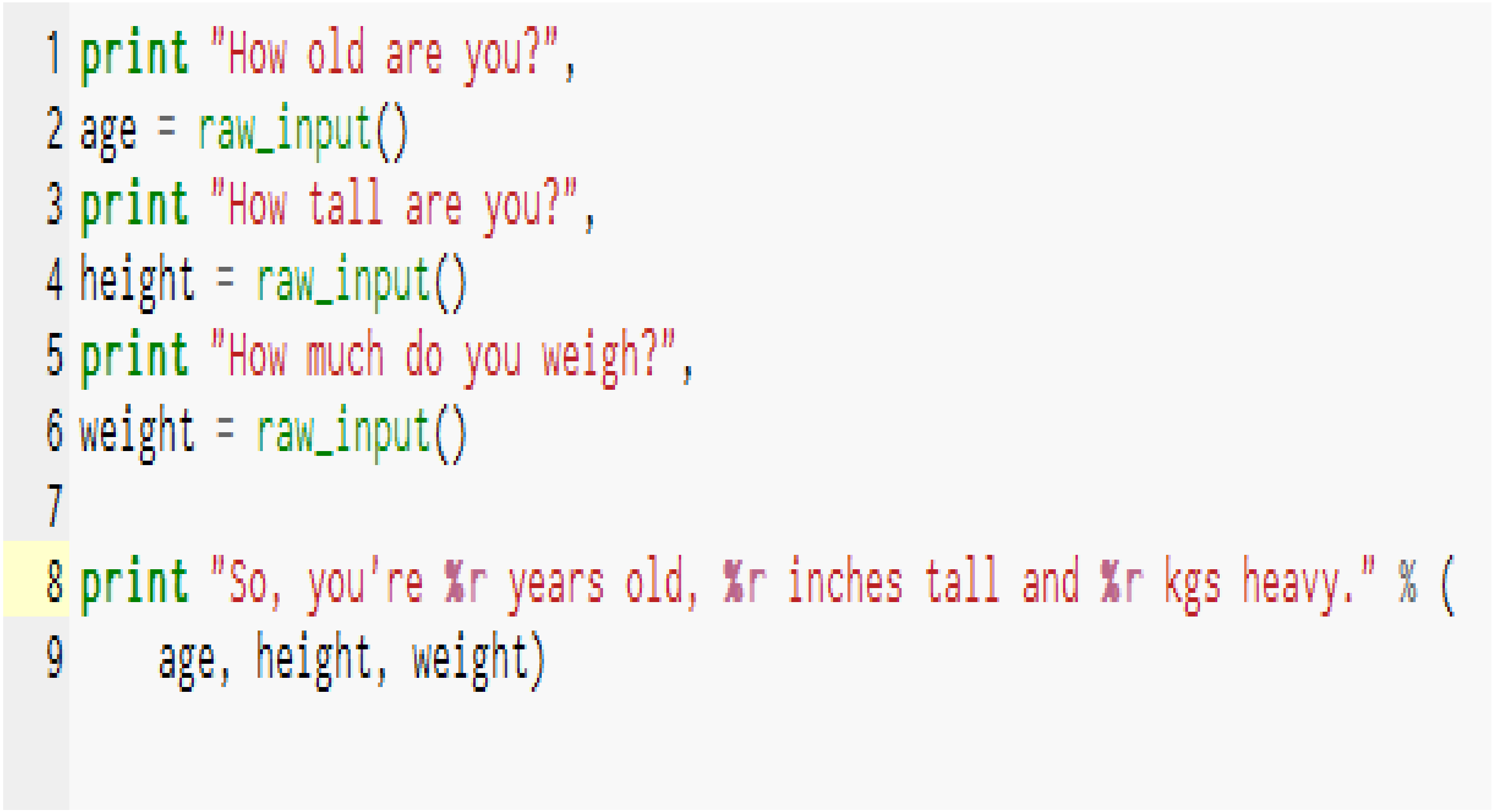
4. Rewrite the code below with your personal details and run. You should experiment with the two string formatting characters `%s` and `%d`



```python
my_name = 'Riaz Ahmad'
my_age = 49 # I use oil of ulay!
my_height = 70 # inches
my_weight = 85 # kgs
my_eyes = 'Brown'
my_teeth = 'White'
my_hair = 'Black'

print "Let's talk about %s." % my_name
print "He's %d inches tall." % my_height
print "He's %d pounds heavy." % my_weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
print "His teeth are usually %s depending on the toothpaste." % my_teeth

# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (
    my_age, my_height, my_weight, my_age + my_height + my_weight)
```

Exercise 1.2

5. This exercise is for experimenting with data input from the keyboard. You will also notice a new character `%r`.



```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "How much do you weigh?",
weight = raw_input()

print "So, you're %r years old, %r inches tall and %r kgs heavy." % (
    age, height, weight)
```

Control statements – `Else and If`

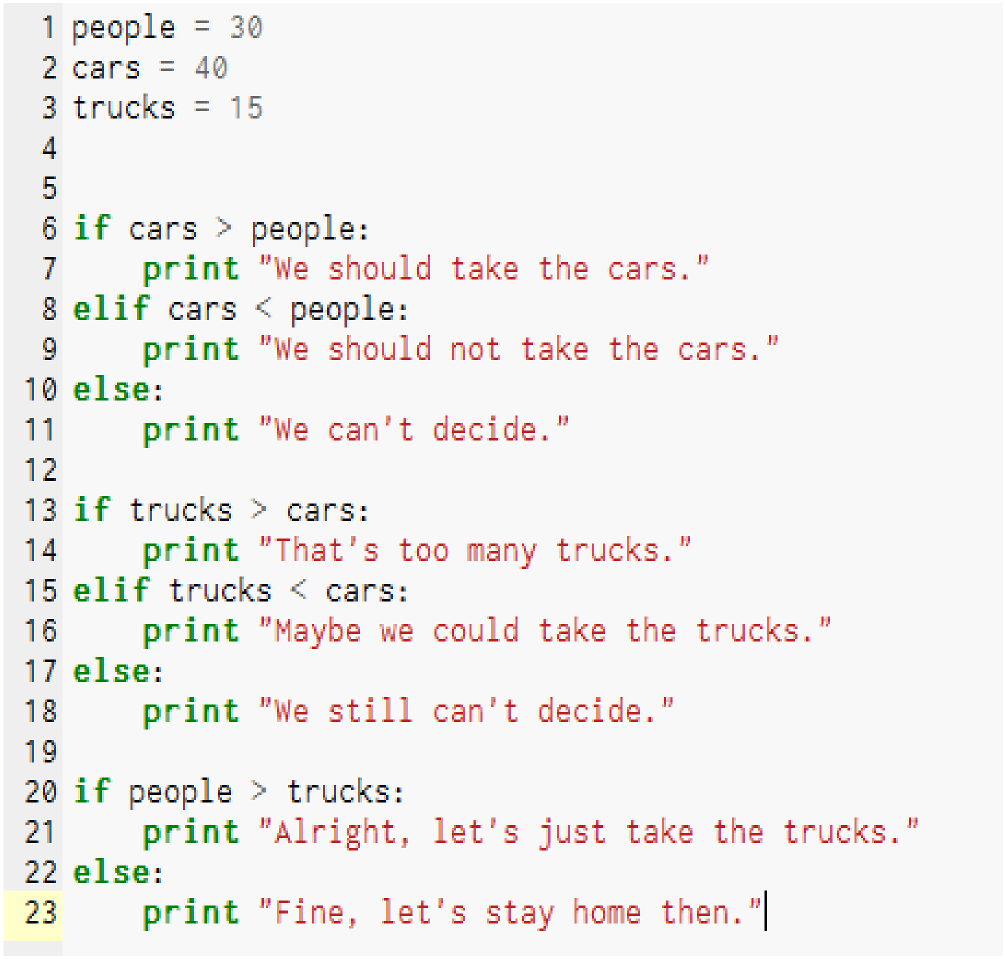
An if-statement creates what is called a "branch" in the code.

The if-statement tells the Python script, “if this boolean expression is True, then run the code under it, otherwise skip it.“

The code under the if needs to be indented four spaces? A colon at the end of a line is how you tell Python you are going to create a new "block" of code, and then indenting four spaces tells Python what lines of code are in that block. This is *exactly* the same thing you will do with functions later. Not indenting will give an error.

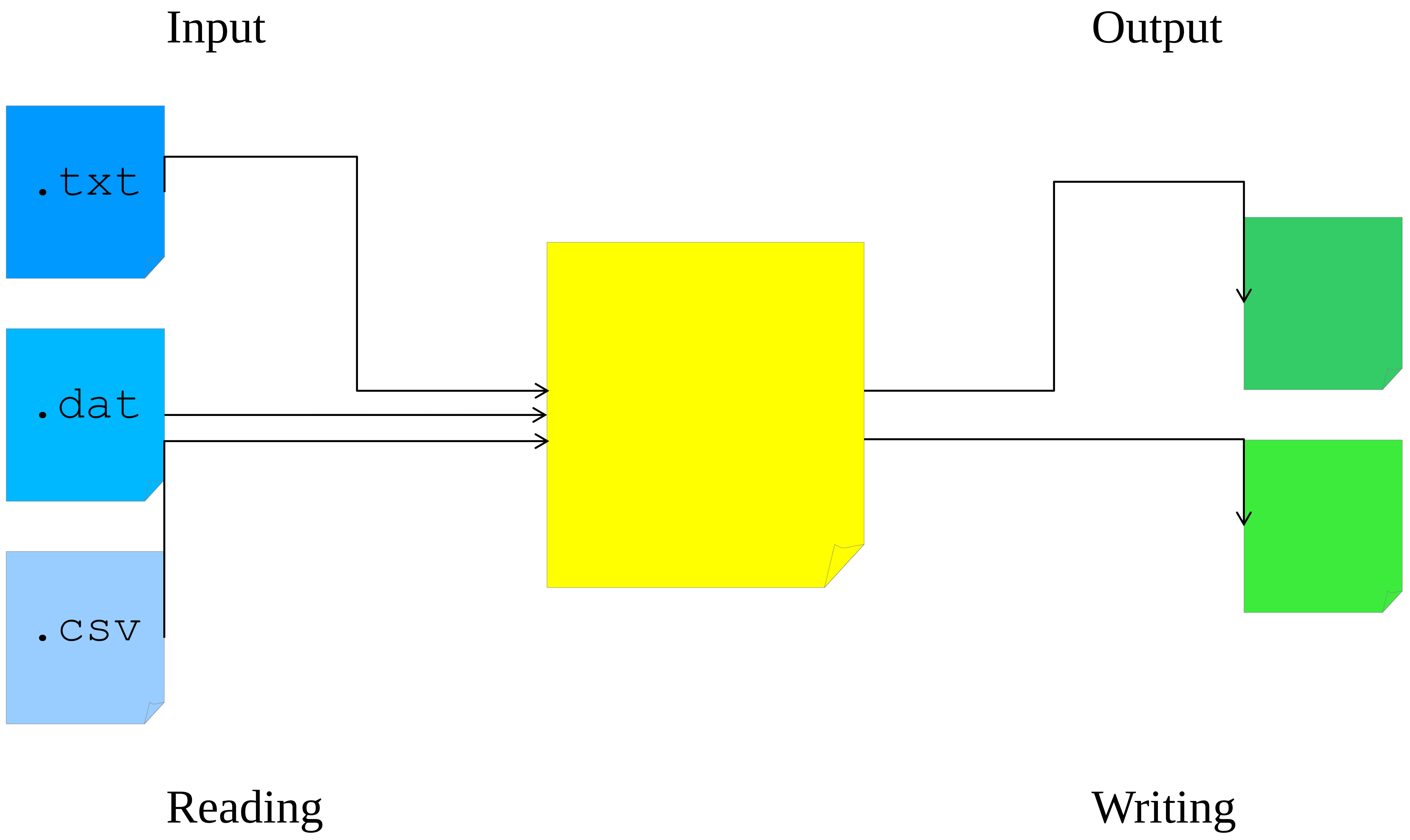
Exercise 2.0

Look at the code below. Make sure you are happy with the logic. Add more Boolean expressions and increase the complexity. Use some of the earlier script to increase complexity.



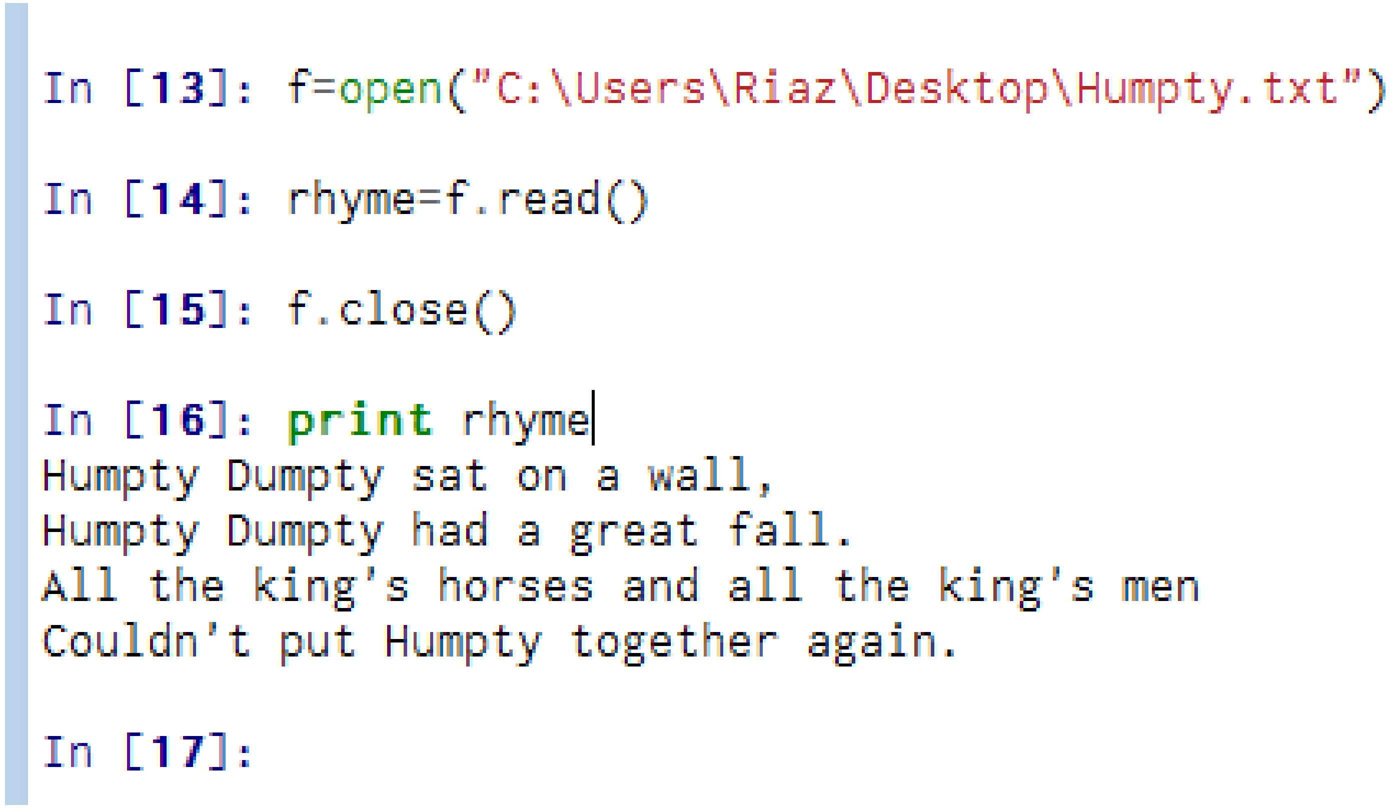
```
people = 30
cars = 40
trucks = 15


if cars > people:
    print "We should take the cars."
elif cars < people:
    print "We should not take the cars."
else:
    print "We can't decide."

if trucks > cars:
    print "That's too many trucks."
elif trucks < cars:
    print "Maybe we could take the trucks."
else:
    print "We still can't decide."

if people > trucks:
    print "Alright, let's just take the trucks."
else:
    print "Fine, let's stay home then."
```

Files



Reading text files

This is the simplest script that opens a file for reading, loads its contents into a text variable and closes the file. In my example the file called Humpty.txt has location C:\Users\Riaz\Desktop\Humpty.txt

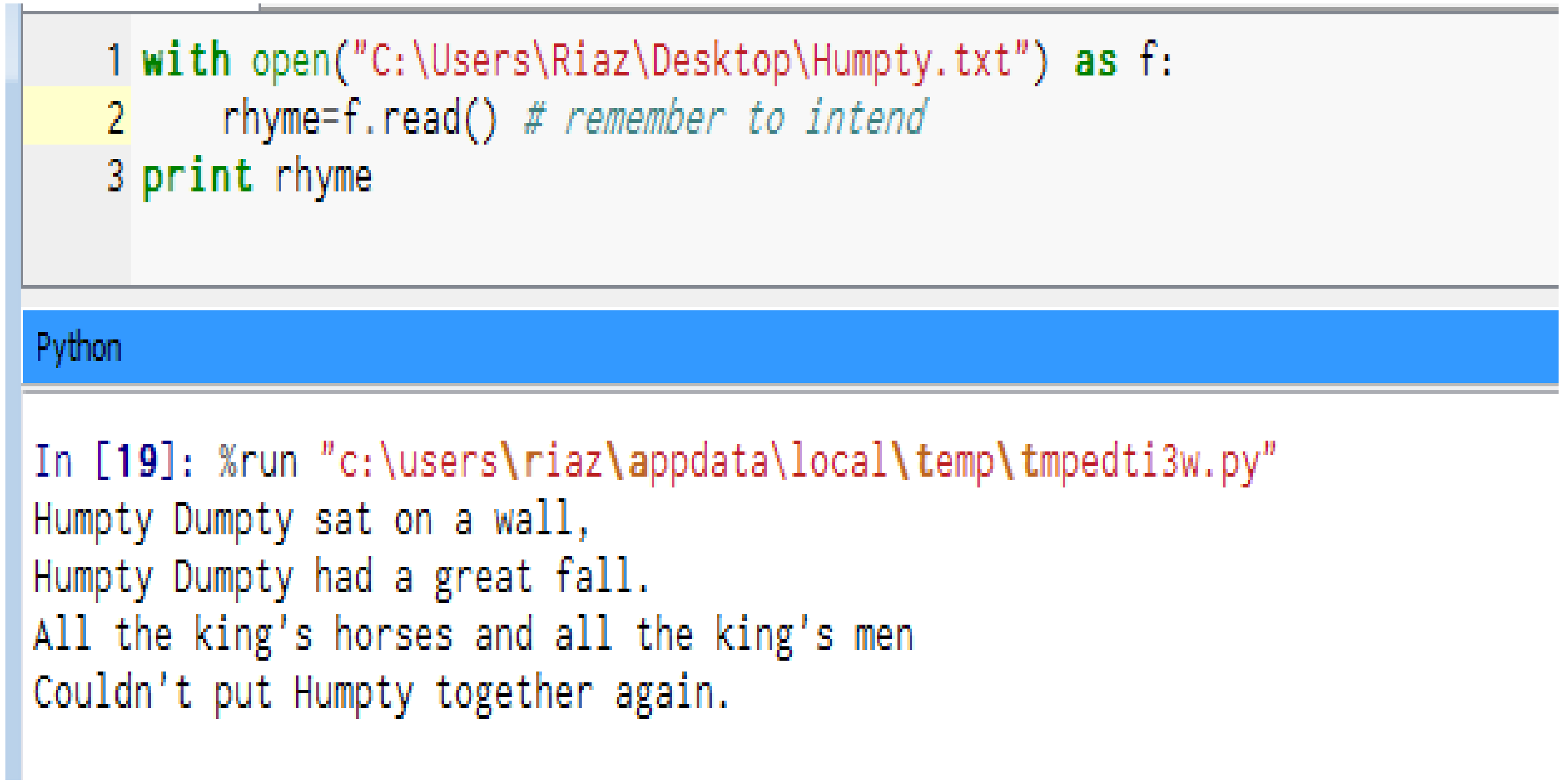


```
In [13]: f=open("C:\Users\Riaz\Desktop\Humpty.txt")

In [14]: rhyme=f.read()

In [15]: f.close()

In [16]: print rhyme
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.

In [17]:
```

Single quotes also work

The `with` statement

Closing a file is easily overlooked. For this and other reasons it is better to use the **with** statement:



```python
with open("C:\Users\Riaz\Desktop\Humpty.txt") as f:
    rhyme=f.read() # remember to intend
print rhyme
```

Python

```
In [19]: %run "c:\users\riaz\appdata\local\temp\tmpedti3w.py"
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```

Writing to files

- Opening files for writing:

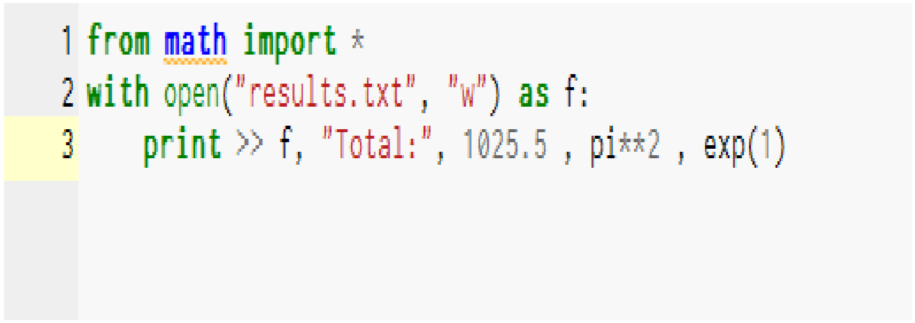
  1. First decide what to do if there is already a files with the same name.
  2. If you want to delete the file and start from scratch, pass `“w”` as the second argument to `open()`. Example: `open(“myfile.txt”, “w”)`
  3. If you want to append text to that file, pass “a” as the second argument to open(). Example: open(“myfile.txt”, “a”)

- Writing data:

print has a special syntax causing its output to be redirected to a file:

```
print >> file_object, comma_separated_expressions
```

Example:

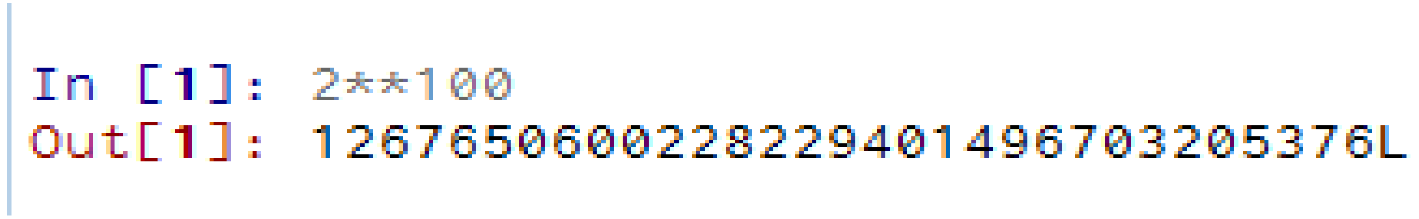


```
from math import *
with open("results.txt", "w") as f:
    print >> f, "Total:", 1025.5 , pi**2 , exp(1)
```

Numbers in Python

A number can be represented by different types of variables:

- int for integer
- long integers for integer not in the previous interval. They are stored in a more complex format. Such integers are printed out with an L at the end:



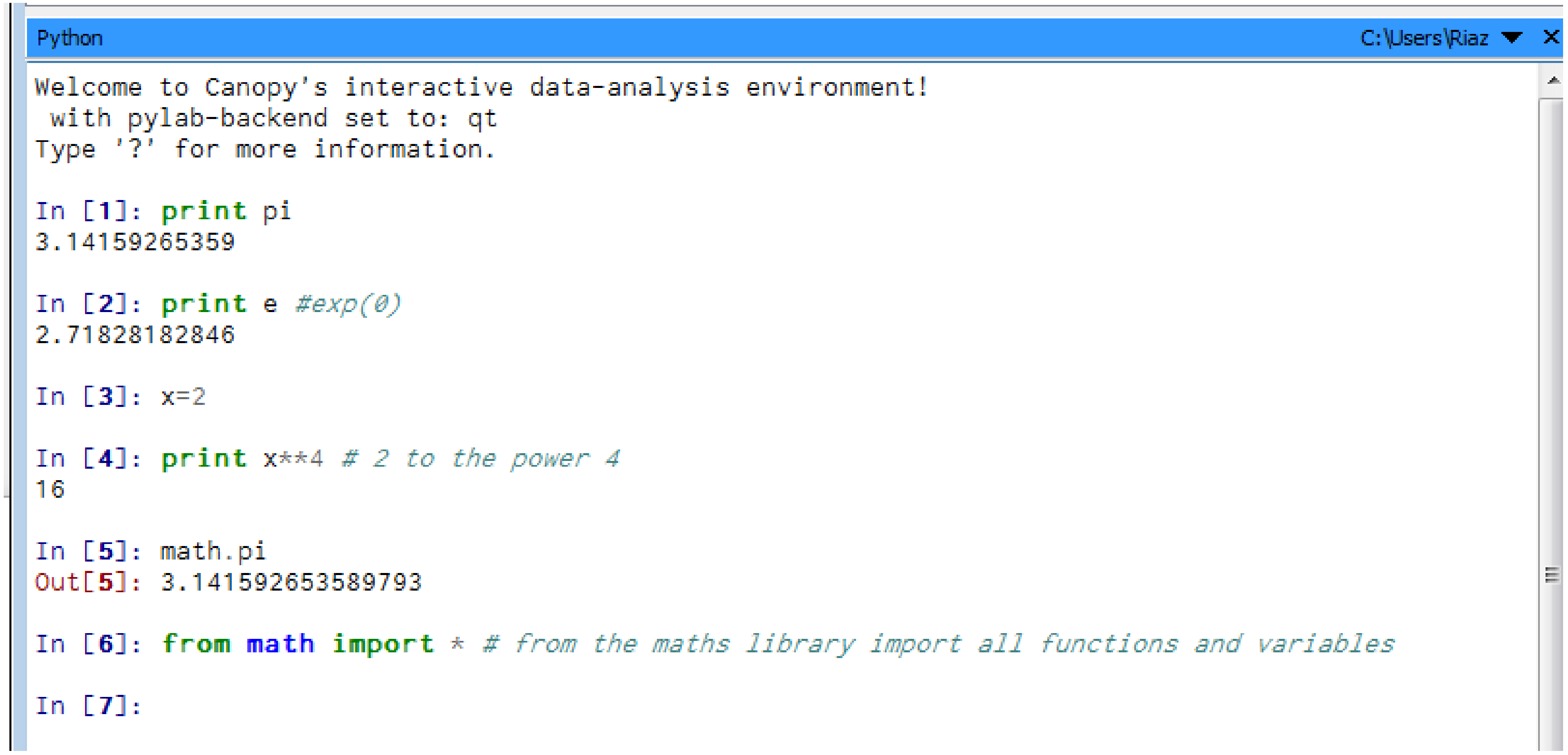
```
In [1]: 2**100
Out[1]: 1267650600228229401496703205376L
```

The range of these long integers is only limited by the amount of available memory in the computer. Operations on long integers are slower than on normal integers.

Simple Arithmetic Calculations – BIDMAS or BODMAS

$$\begin{tikzcd}
\text{✗} & 4 + 8 / 2 \arrow[dl] \arrow[dr] & \text{✓} \\
(4 + 8) / 2 \arrow[d] & & 4 + (8 / 2) \arrow[d] \\
6 & & 8
\end{tikzcd}$$

Manipulating in-built functions



```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]: print pi
3.14159265359

In [2]: print e #exp(0)
2.71828182846

In [3]: x=2

In [4]: print x**4 # 2 to the power 4
16

In [5]: math.pi
Out[5]: 3.141592653589793

In [6]: from math import * # from the maths library import all functions and variables

In [7]:
```

Converting text to floats

```
>>> float('10.0')
10.0
```

'10.0' is a string

10.0 is a floating point number

```
>>> float('␣10.␣')
10.0
```

Spaces are ignored

Converting between ints and floats

```
>>> float(10)
10.0

>>> int(10.9)
10

>>> int(-10.9)
-10
```

Truncates fractional part

Converting into text

```
>>> str(10)
'10'
```

$\text{integer} \longrightarrow \text{string}$

```
>>> str(10.000)
'10.0'
```

$\text{float} \longrightarrow \text{string}$

Converting between types

`int()`	anything $\longrightarrow$ integer
`float()`	anything $\longrightarrow$ float
`str()`	anything $\longrightarrow$ string

Functions named after the type they convert *into*.

Exercise 4.0

Write python programs to do the following:

1. Prompt the user with the text “How much?␣”.
2. Convert the user’s answer to a floating point number.
3. Print 2.5 plus that number.

Integer arithmetic

```
>>> 10+5
15
```

```
>>> 25␣-␣5
10
```

Spaces around the operator don’t matter.

```
>>> 20␣*␣5
100
```

```
>>> 20␣/␣5
4
```

```
>>> 20␣/␣3
6
```

Type-casting

```
>>> 20.0/3
6.666666666666667
```

```
>>> float(20)/3
6.666666666666667
```

```
>>> 20/float(3)
6.666666666666667
```

```
>>> float(20/3)
6.0
```

Integer powers **

We wish to calculate 4^3

```
>>> 4␣**␣3
64
```

Spaces *around* the operator don't matter.

```
>>> 4*␣*3
SyntaxError: invalid syntax
```

Spaces *in* the operator do!

Integer remainders

Use “%” to obtain integer remainders

```
>>> 4␣%␣2
0
```

```
>>> 5␣%␣2
1
```

In the example above “%” is modulo and can be used to determine if an integer is even or odd. $x \in \mathbb{Z}$ and $x\%2 = 0$ then x is even, else x is odd.

```
>>> 20%6
3
```

```
>>> -5␣%␣2
1
```

Remainder is always non-negative

How big can a Python integer be?

```
>>> 2**2
4

>>> 4**2
16

>>> 16**2
256

>>> 256**2
65536

>>> 65536**2
4294967296
```

How big can a Python integer be?

```
>>> 4294967296**2
18446744073709551616

>>> 18446744073709551616**2
340282366920938463463374607431768211456

>>> 340282366920938463463374607431768211456**2
1157920892373161954235709850086879078532699846
65640564039457584007913129639936

>>> 1157920892373161954235709850086879078532699
84665640564039457584007913129639936**2
1340780792994259709957402499820584612747936582
0592393377723561443721764030073546976801874298
1669034276900318581864860508537538828119465699
46433649006084096
```

How big can a Python integer be?

1044388881413152506691752710716624382579964249047383780384233483283 9
5390797155745684882681193499755834089010671443926283798757343818579 3
6072632360878513652779459569765437099983403615901343837183144280700 1
1855946226376318839397712745672334684344586617496807908705803704071 2
8404874011860911446797778359802900668693897688178778594690563019026 0
9405995794534328234 169383555
9885291486318237914 084170616
7509366833385055103 825837188
0918336567512213184 449219461
7023806505913245610825731835380087608622102834270197698202313169017 6
7800667519548507992163641937028537512478401490715913545998279051339 9
61155179427110683113409058427288427979155484978295432353 45
9061394905987693002122963395687782878948440616007412945674
7164237715481632138063104590291613692670834285644073044789
4657634732238502672530598997959960907994692017746248177184
9250178329070473119433165550807568221846571746373296884912
Except for machine memory
5700244092661691087414838507841192980452298185733897764810312608590 3
0013024134671897266732164915111316029207817380334360902438047083404 0
3154190336

There is no limit!

Floating point numbers

$\mathbb{R}$

1.0

0.33333333

3.14159265

2.71828182

Basic operations

```
>>> 20.0 + 5.0
25.0
```

```
>>> 20.0 - 5.0
15.0
```

```
>>> 20.0 * 5.0
100.0
```

```
>>> 20.0 / 5.0
4.0
```

```
>>> 20.0 ** 5.0
3200000.0
```

Equivalent to integer arithmetic

Floating point imprecision

```
>>> 1.0 / 3.0
0.3333333333333333


>>> 10.0 / 3.0
3.3333333333333335
```

If you are relying on this last decimal place, you are doing it wrong!

≈ **17** significant figures

Hidden imprecision

```
>>> 0.1
0.1

>>> 0.1 + 0.1
0.2

>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Really: if you are relying on this last decimal place, you are doing it wrong!

How big can a Python float be? — 1

```
>>> 65536.0**2
4294967296.0
```

So far, so good.

```
>>> 4294967296.0**2
1.8446744073709552e+19
```

Switch to “scientific notation”

1.8446744073709552 e+19

$1.8446744073709552 \times 10^{19}$

Floats are not exact

```
>>> 4294967296.0**2
1.8446744073709552e+19
```

Floating point

```
>>> 4294967296**2
18446744073709551616
```

Integer

$$1.8446744073709552{\times}10^{19} \longrightarrow 18446744073709552000$$

$$\begin{array}{rr} & 18446744073709552000 \\ - & 18446744073709551616 \\ \hline & 384 \end{array}$$

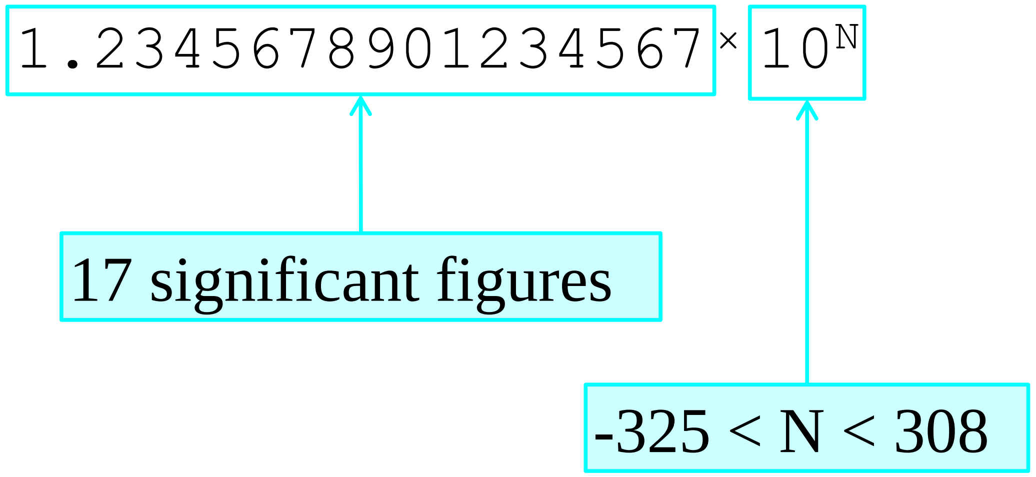
How big can a Python float be? — 2

```
>>> 1.8446744073709552e+19**2
3.402823669209385e+38

>>> 3.402823669209385e+38**2
1.157920892373162e+77

>>> 1.157920892373162e+77**2
1.3407807929942597e+154

>>> 1.3407807929942597e+154**2
OverflowError: (34,
'Numerical result out of range')
```

So far, so good.

Too big!

Floating point limits



Positive values:

$4.94065645841 \times 10^{-324} < N < 8.98846567431 \times 10^{307}$

Scientific notation

Scientific notation is a convenient way of expressing very large or very small numbers.

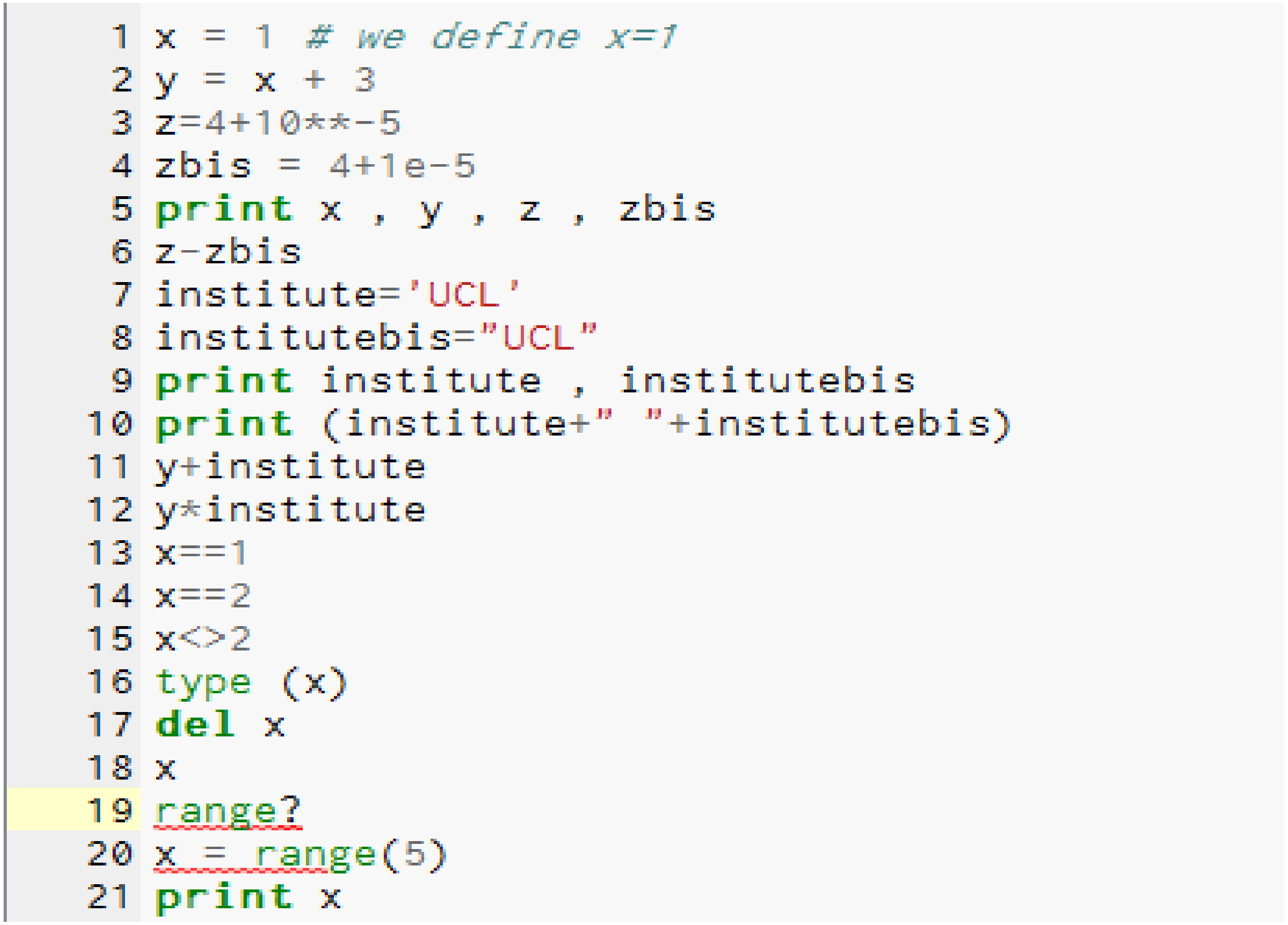
General form:

$aen = a \times 10^n$, where a is a float and n is an integer.

Examples:

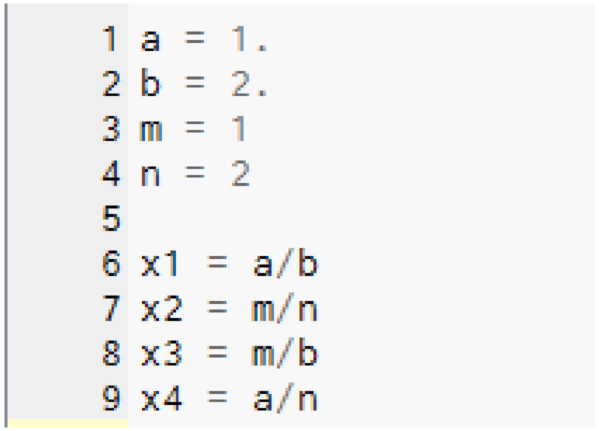
$$3e8 = 3 \times 10^8$$
$$9.109e - 19 = 9.109 \times 10^{-19}$$

Exercise: In the Interactive Python Terminal, try these instructions and write what each instruction does. First example already done



```
x = 1 # we define x=1
y = x + 3
z=4+10**-5
zbis = 4+1e-5
print x , y , z , zbis
z-zbis
institute='UCL'
institutebis="UCL"
print institute , institutebis
print (institute+" "+institutebis)
y+institute
y*institute
x==1
x==2
x<>2
type (x)
del x
x
range?
x = range(5)
print x
```

Exercise: Write this small program in a script and write in comment what will be the type and the value of the variables x_1; x_2; x_3 and x_4 after executing this code:

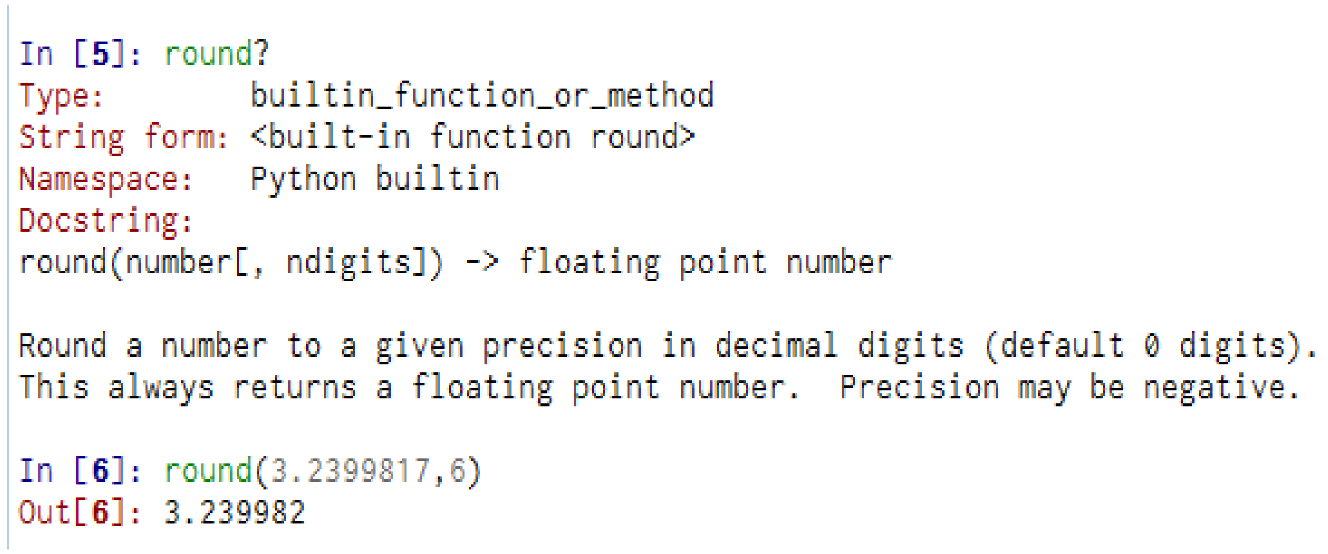


```
a = 1.
b = 2.
m = 1
n = 2

x1 = a/b
x2 = m/n
x3 = m/b
x4 = a/n
```

Validate your results using the instruction `type`(variable)

More notes on integers

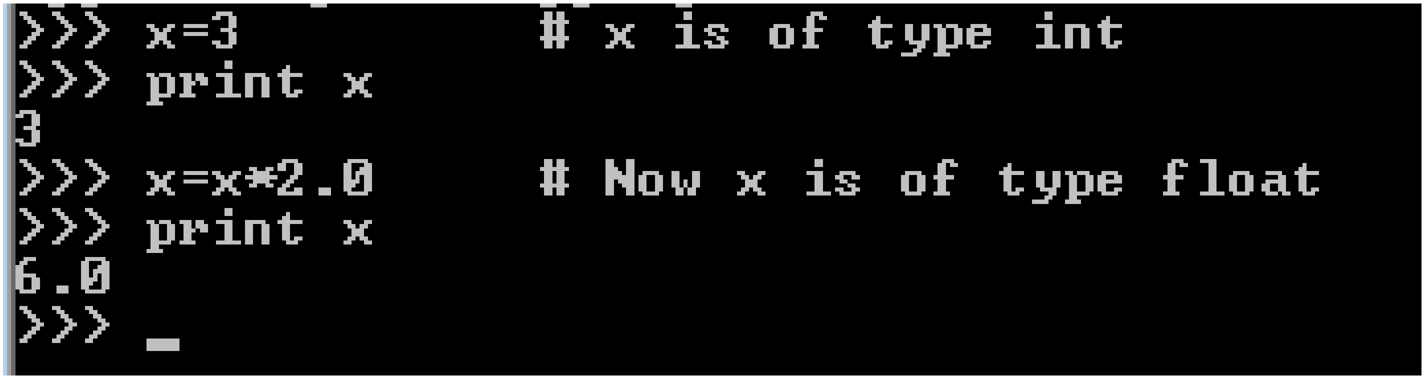
```
In [1]:int(3.4)
Out[1]:3
In [2]:int(3.9)
Out[2]:3
In [3]:int(round(3.4))
Out[3]:3
In [4]:int(round(3.9))
Out[4]:4
```



```
In [5]: round?
Type:        builtin_function_or_method
String form: <built-in function round>
Namespace:   Python builtin
Docstring:
round(number[, ndigits]) -> floating point number

Round a number to a given precision in decimal digits (default 0 digits).
This always returns a floating point number.  Precision may be negative.

In [6]: round(3.2399817,6)
Out[6]: 3.239982
```

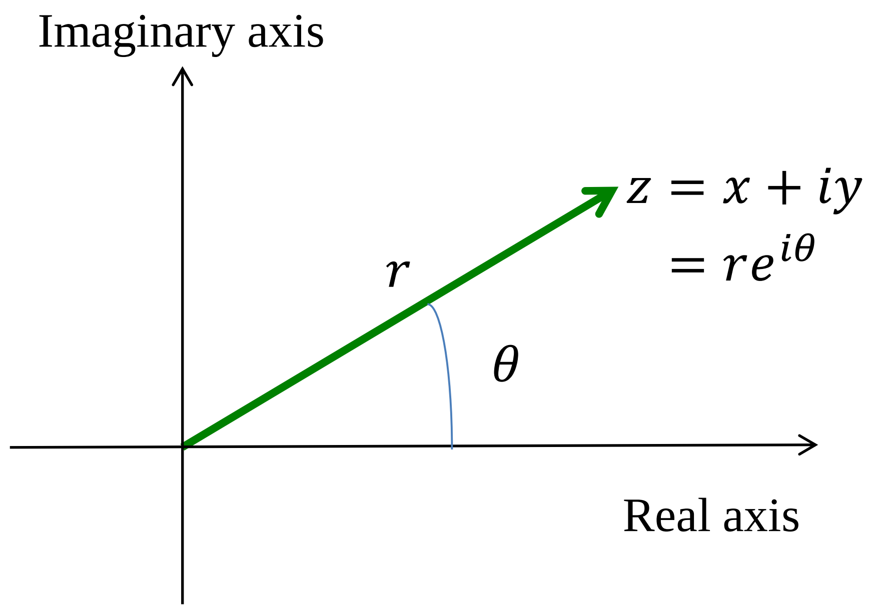
Variables

In programming languages generally, a variable name represents a value of a given type (int, float, etc.) stored in a fixed memory location. The value of a variable can be changed but not the variable type. This is not the case in Python where variables are *typed dynamically* as illustrated below

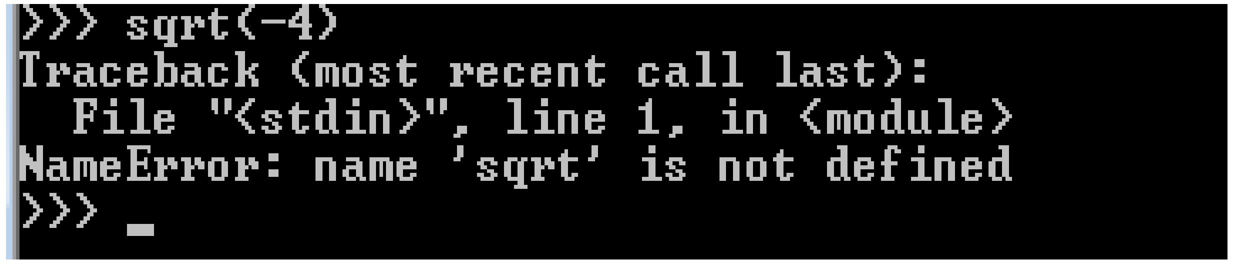


```
>>> x=3          # x is of type int
>>> print x
3
>>> x=x*2.0      # Now x is of type float
>>> print x
6.0
>>> _
```

Complex numbers ℂ



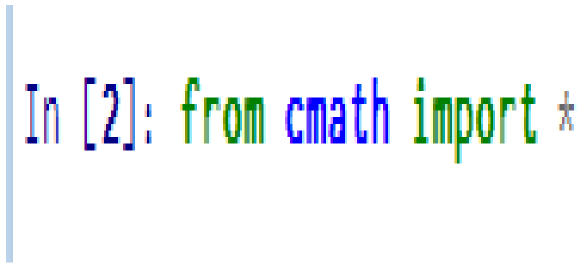
Python has all the functions to manipulate complex numbers.
In Python $j = \sqrt{-1}$



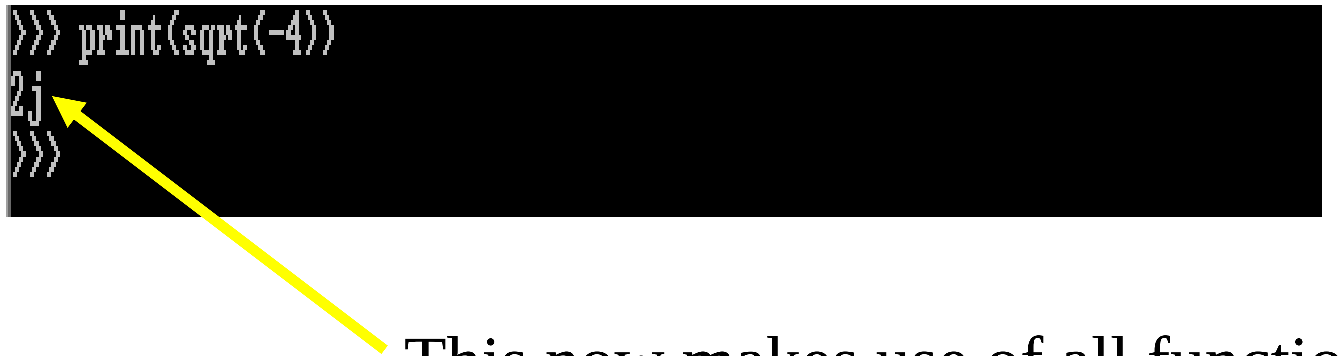
```
>>> sqrt(-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> _
```

$$z = Re(z) + Im(z)j$$

cmath library



```
In [2]: from cmath import *
```

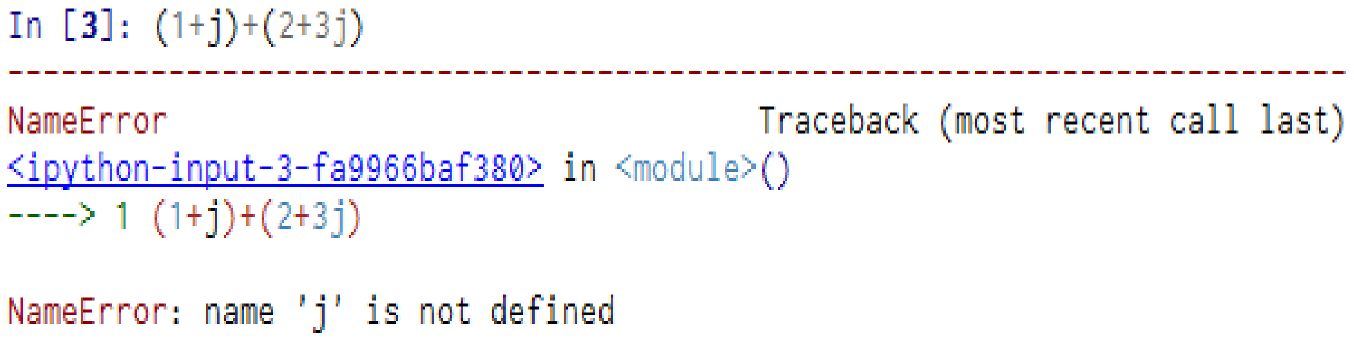


```
>>> print(sqrt(-4))
2j
>>>
```

This now makes use of all functions related to complex numbers. Gives the correct value of $-2i$

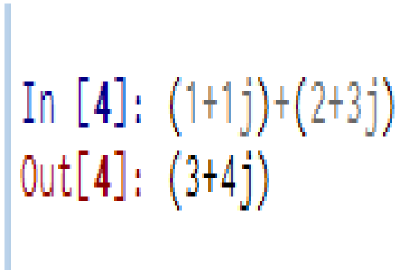
Calculations in ℂ

We now have all the functions to manipulate complex numbers.



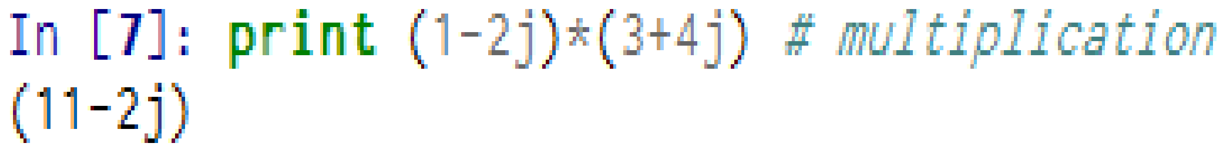
```
In [3]: (1+j)+(2+3j)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-fa9966baf380> in <module>()
----> 1 (1+j)+(2+3j)

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

Now try the following

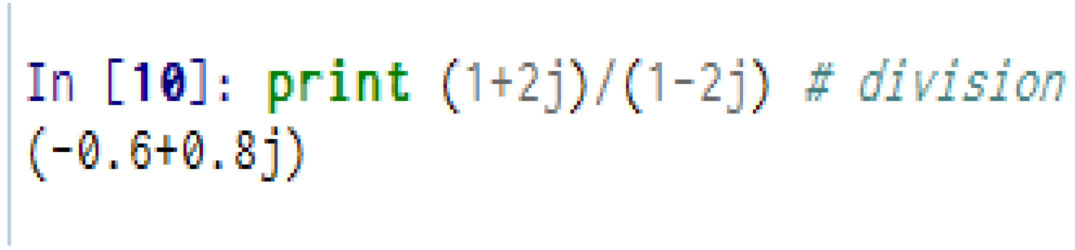


```
In [4]: (1+1j)+(2+3j)
Out[4]: (3+4j)
```

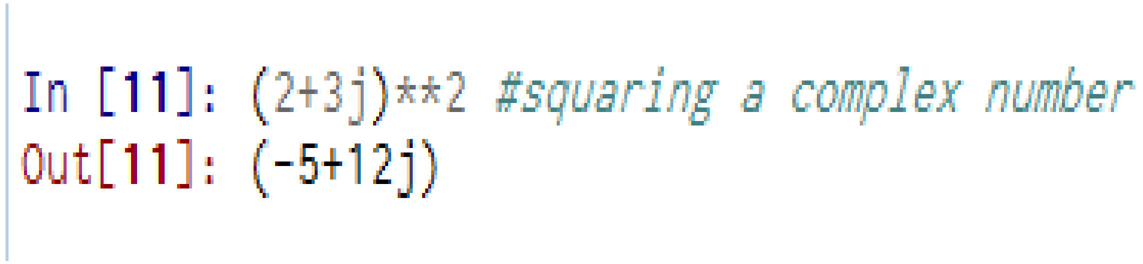
ℂ Operations in Python



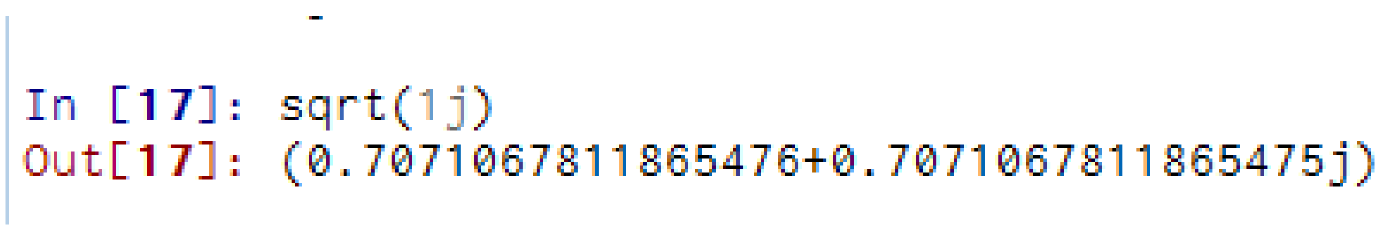
```
In [7]: print (1-2j)*(3+4j) # multiplication
(11-2j)
```



```
In [10]: print (1+2j)/(1-2j) # division
(-0.6+0.8j)
```



```
In [11]: (2+3j)**2 #squaring a complex number
Out[11]: (-5+12j)
```



```
In [17]: sqrt(1j)
Out[17]: (0.7071067811865476+0.7071067811865475j)
```

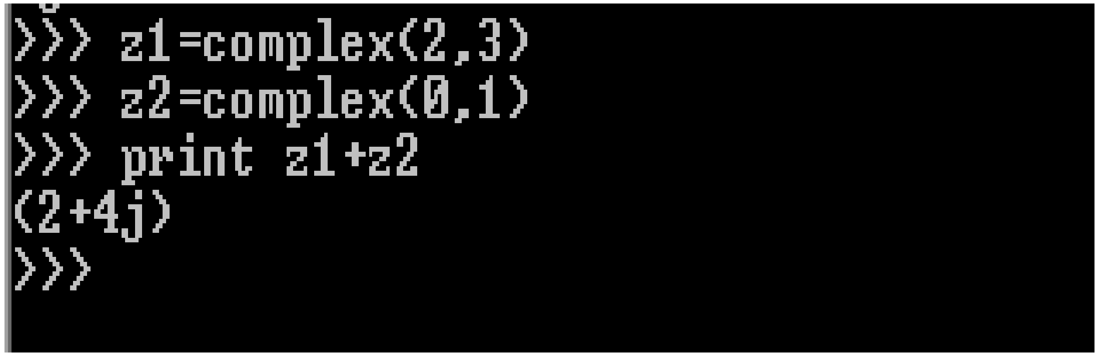
Other ways of handling $\mathbb{C}$

Complex number $z = x + iy$ can also be expressed as a two-tuple $z = (x, y)$ using `complex(x,y)`

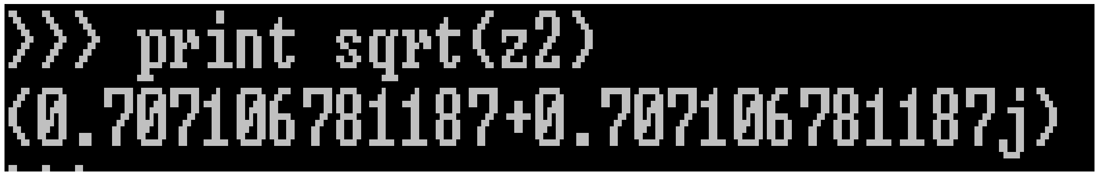
$z_1 = 2 + 3i$ is written as `complex(2,3)`

$z_2 = i$ is written as `complex(0,1)`

$z_3 = 1$ is written as `complex(1,0)`



```
>>> z1=complex(2,3)
>>> z2=complex(0,1)
>>> print z1+z2
(2+4j)
>>>
```



```
>>> print sqrt(z2)
(0.707106781187+0.707106781187j)
```

More operations in ℂ -1

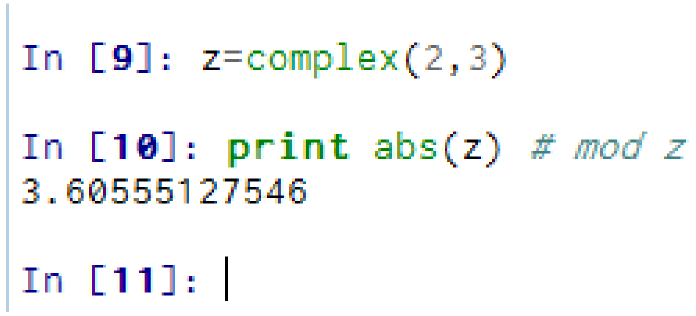
The length of z is the modulus given by

$$|z| = \sqrt{x^2 + y^2}$$

In Python the one parameter, absolute function written

```
abs(..)
```

gives the modulus of a complex number.



```
In [9]: z=complex(2,3)

In [10]: print abs(z) # mod z
3.60555127546

In [11]: 
```

More operations in ℂ -2

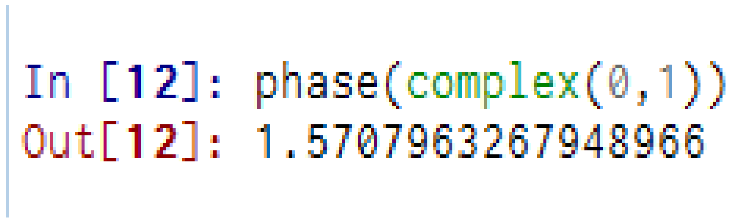
The argument of z, written $arg(z)$ is the angle between $z(x,y)$ and the real axis where

$$arg(z) = arctan\left(\frac{y}{x}\right)$$

The one parameter function

`phase(..)`

gives the argument in radians. The principal value is given, i.e. $-\pi \leq arg(z) \leq \pi$.



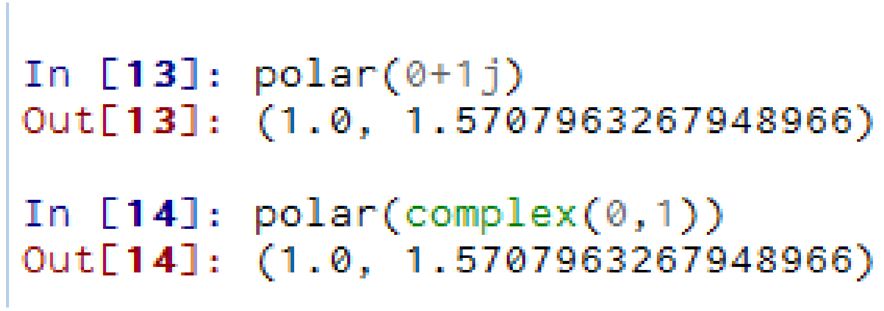
```
In [12]: phase(complex(0,1))
Out[12]: 1.5707963267948966
```

More operations in $\mathbb{C}$ -3

Converting complex numbers from Cartesian (x, y) to polar form (r, θ) is a fairly straightforward mathematical exercise. How is this done in Python? A single argument function

`polar(..)` $\longrightarrow (r, \theta);\ \ r = |z|$ and $\theta = tan^{-1}(y/x)$

- `polar(x+yj)`
- `polar(complex(..,..))`



```
In [13]: polar(0+1j)
Out[13]: (1.0, 1.5707963267948966)

In [14]: polar(complex(0,1))
Out[14]: (1.0, 1.5707963267948966)
```

Exercise

Evaluate and print out the following calculations:

1. $223 \div 71$
2. $(1 + 1/10)^{10}$
3. $(1 + 1/100)^{100}$
4. $(1 + 1/1000)^{1000}$

Comparisons – Truth and Falsehood

```
>>>   6 < 10
True
```

Asking the question

✓

```
>>>   6 > 10
False
```

Asking the question

✗

```
>>>   5 == 10
False
```

5 equal to 10?

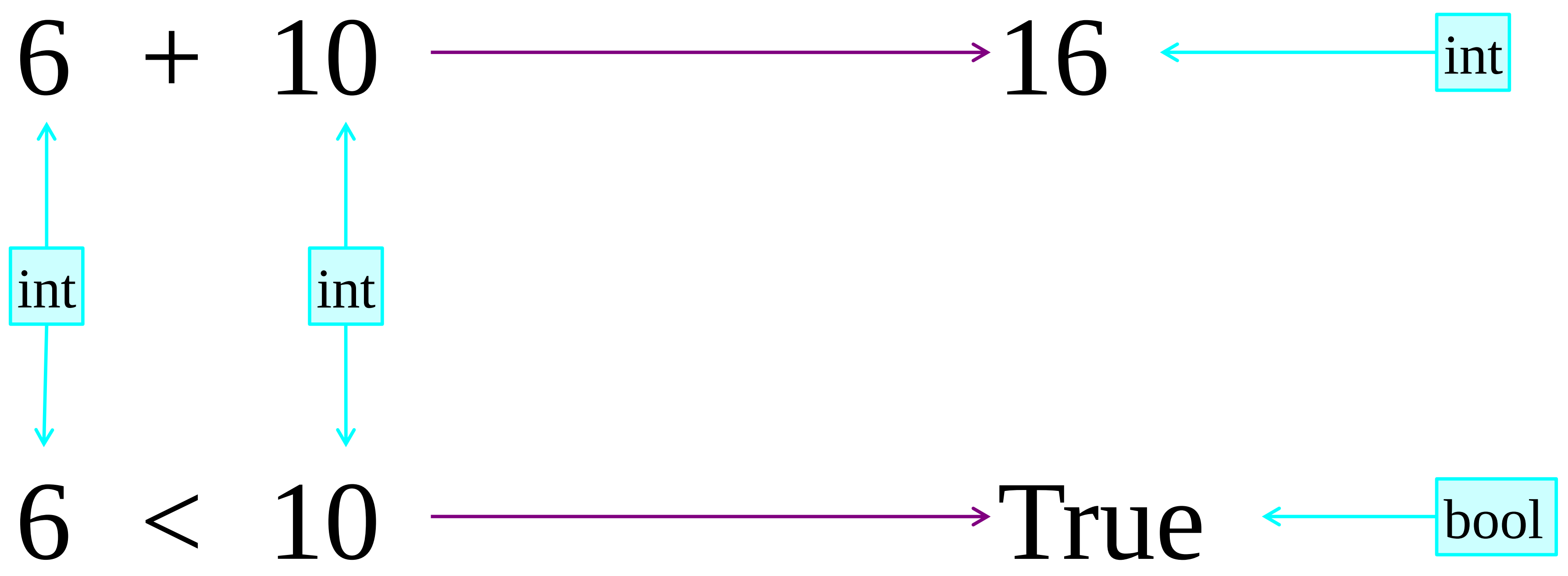
```
>>>   6 <> 10
True
```

6 is not equal to 10?

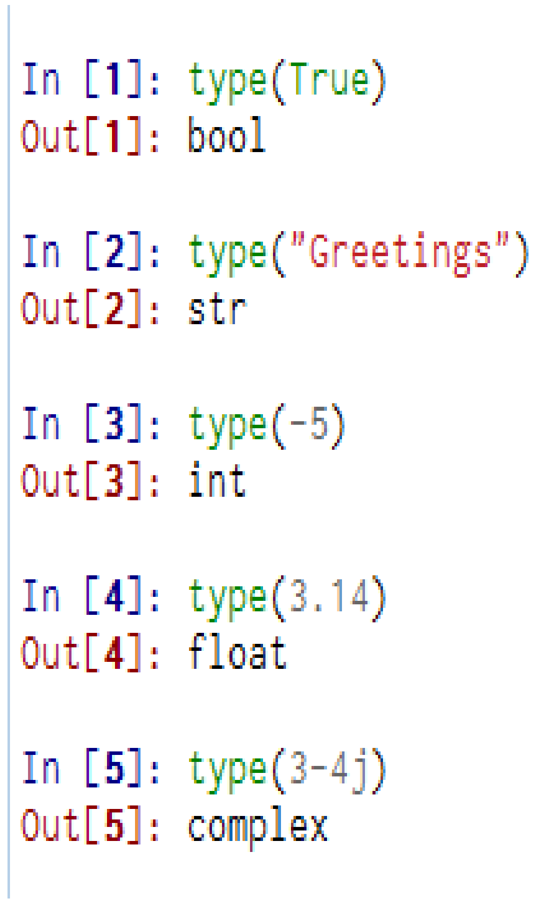
True & False

```
>>>   type(True)
<type 'bool'>
```

“Booleans”

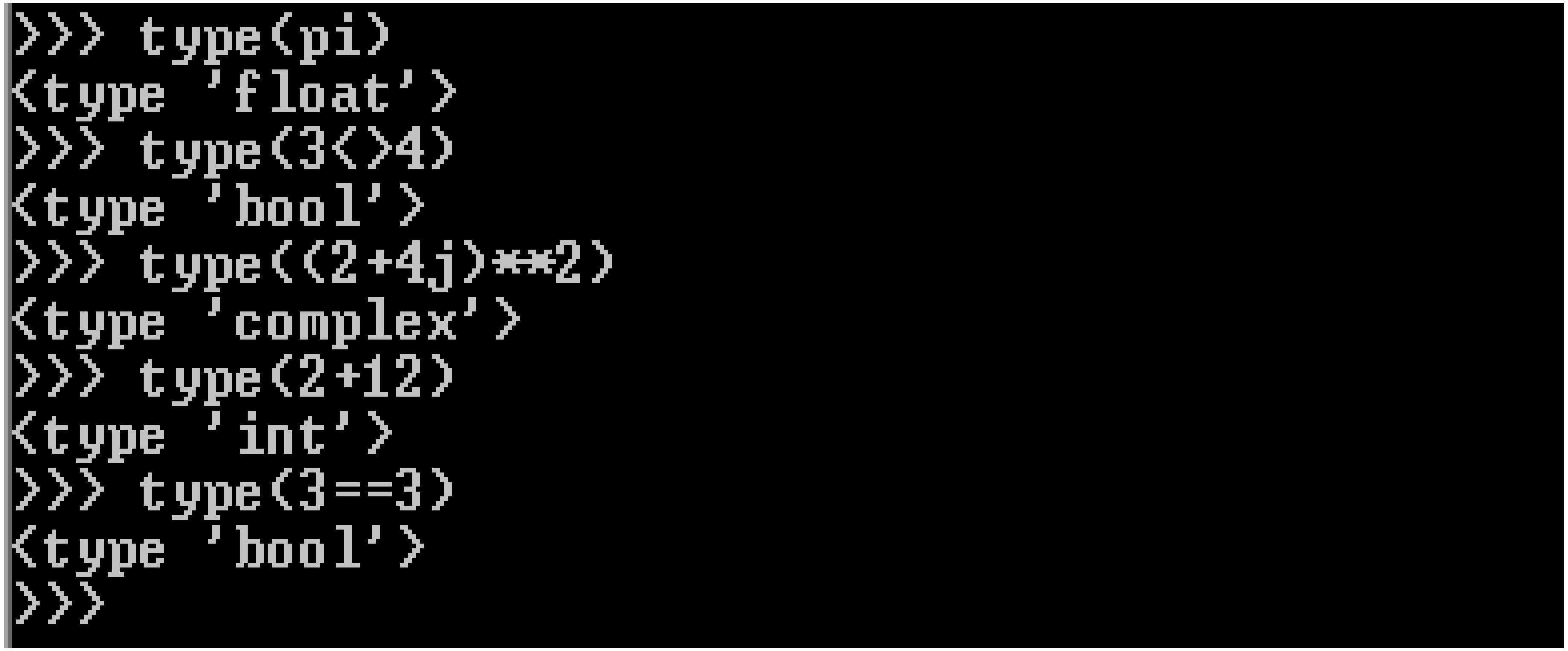


Check on type



```
In [1]: type(True)
Out[1]: bool

In [2]: type("Greetings")
Out[2]: str

In [3]: type(-5)
Out[3]: int

In [4]: type(3.14)
Out[4]: float

In [5]: type(3-4j)
Out[5]: complex
```

Examples



```
>>> type(pi)
<type 'float'>
>>> type(3<>4)
<type 'bool'>
>>> type((2+4j)**2)
<type 'complex'>
>>> type(2+12)
<type 'int'>
>>> type(3==3)
<type 'bool'>
>>>
```

True & False

bool	`True`	✓
bool	`False`	✗

The Boolean type has precisely two values

Six comparison operators

Maths	Python	Meaning
$=$	`==`	equals
$\neq$	`!=` `<>`	not equal to
$<$	`<`	less than
$>$	`>`	greater than
$\leq$	`<=`	less than or equal
$\geq$	`>=`	greater than or equal

“Syntactic sugar”

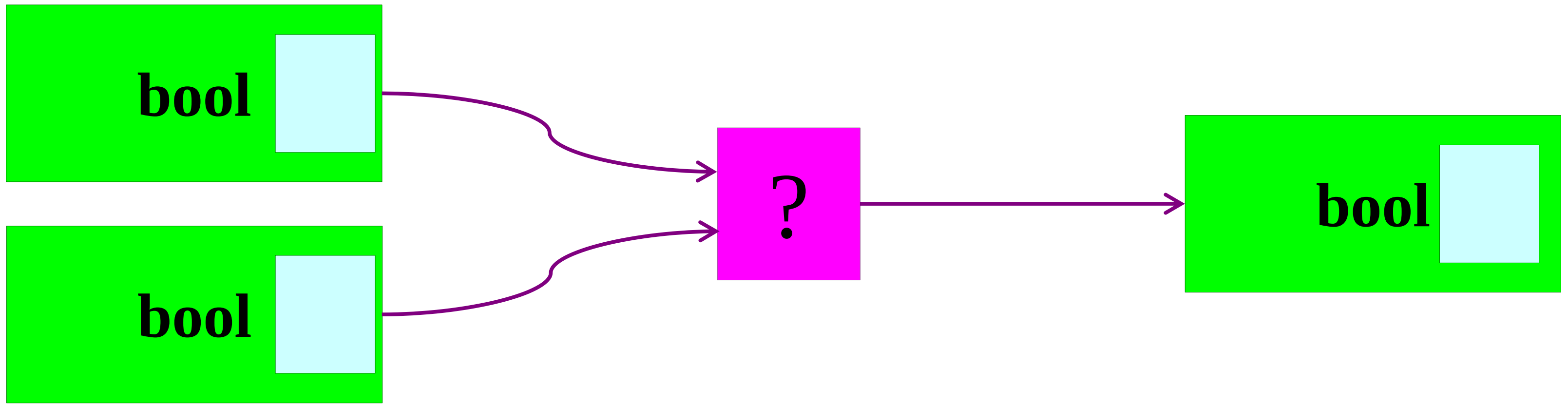
A common question in maths: $x \in [a, b]$?

`0 < number < 10` $\longrightarrow$

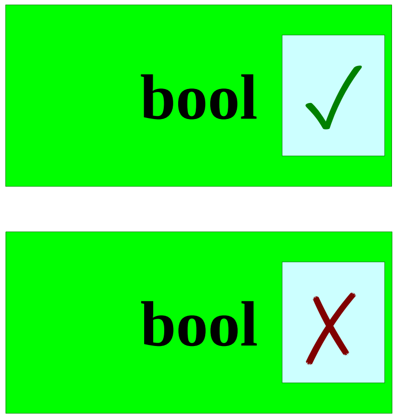
```
0 < number
   and
number < 10
```

```
>>> number = 4
>>> 0 < number < 10
True
```

Boolean operations

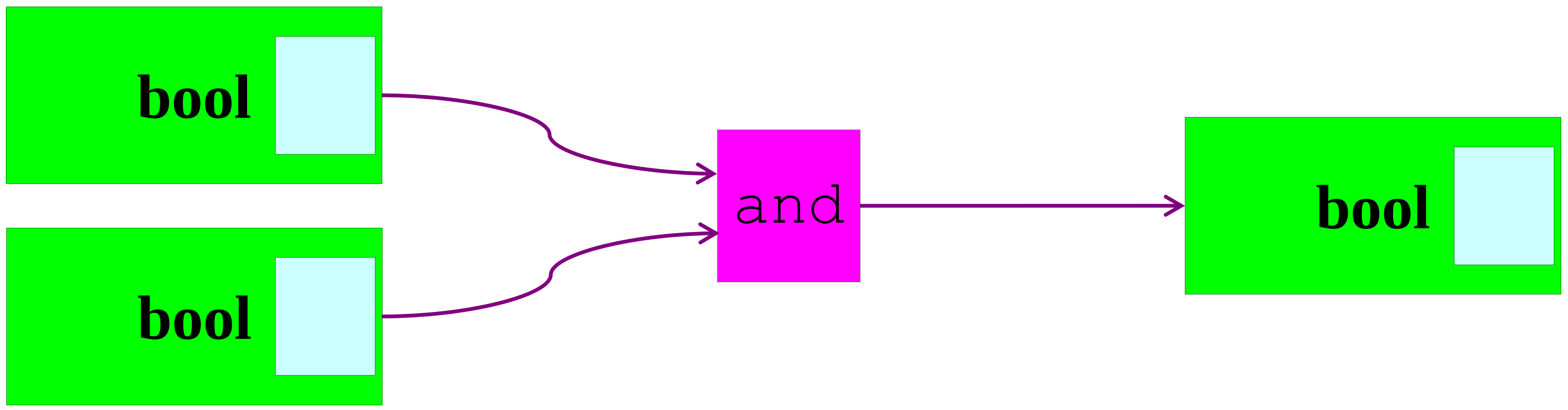


Numbers have arithmetic operations +, –, * …



What operations do Booleans have?

Boolean operations — “and”



```
True  and True   ⟶   True
True  and False  ⟶   False
False and True   ⟶   False
False and False  ⟶   False
```

Both have to be True

and is a strong condition

Boolean operations — “and”

Examples:

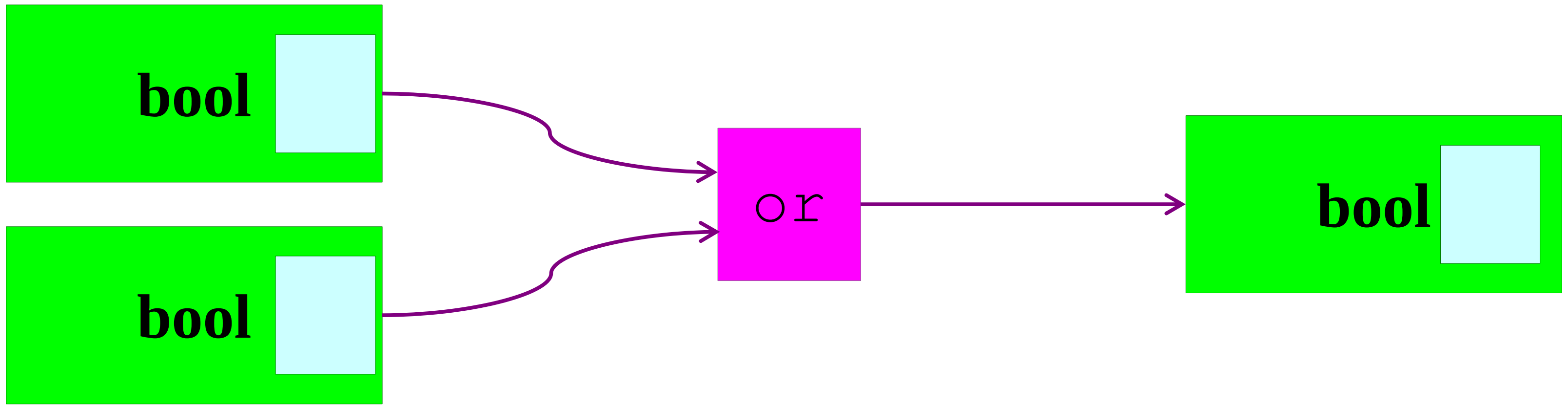
```
>>> 5 < 10 and 6 < 8
True
```

$$\begin{tikzcd}
5 < 10 \arrow[r] & \text{True} \arrow[dr, dash] & & \\
 & & \text{and} \arrow[r] & \text{True} \\
6 < 8 \arrow[r] & \text{True} \arrow[ur, dash] & &
\end{tikzcd}$$

```
>>> 5 < 10 and 6 > 8
False
```

$$\begin{tikzcd}
5 < 10 \arrow[r] & \text{True} \arrow[dr, dash] & & \\
 & & \text{and} \arrow[r] & \text{False} \\
6 > 8 \arrow[r] & \text{False} \arrow[ur, dash] & &
\end{tikzcd}$$

Boolean operations — "`or`"



```
True   or  True   ───→  True
True   or  False  ───→  True
False  or  True   ───→  True
False  or  False  ───→  False
```

At least one has to be True

Weaker condition than and

Boolean operations — “`or`”

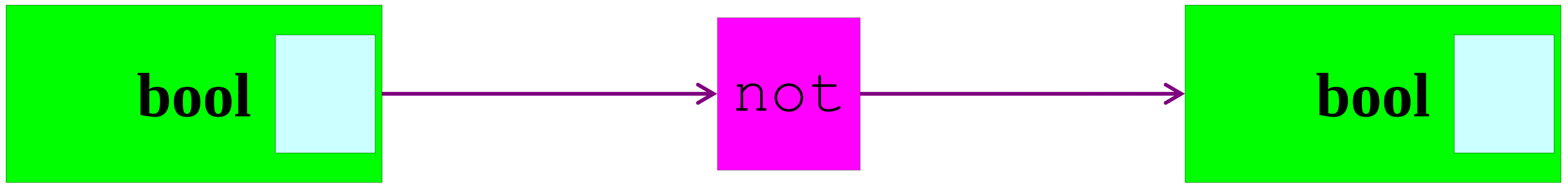
```
>>> 5 < 10 or 6 < 8
True
```

$$\begin{tikzcd} 5 < 10 \arrow[r] & \text{True} \arrow[dr, dash] & & \\ & & \text{or} \arrow[r] & \text{True} \\ 6 < 8 \arrow[r] & \text{True} \arrow[ur, dash] & & \end{tikzcd}$$

```
>>> 5 < 10 or 6 > 8
True
```

$$\begin{tikzcd} 5 < 10 \arrow[r] & \text{True} \arrow[dr, dash] & & \\ & & \text{or} \arrow[r] & \text{True} \\ 6 > 8 \arrow[r] & \text{False} \arrow[ur, dash] & & \end{tikzcd}$$

Boolean operations — “`not`”



```
not True          →          False

not False         →          True
```

Boolean operations — “`not`”

```
>>> not 6 < 7
False
```

$$6 < 7 \longrightarrow \texttt{True}\ \text{—}\texttt{not}\longrightarrow \texttt{False}$$

```
>>> not 6 > 7
True
```

$$6 > 7 \longrightarrow \texttt{False}\text{—}\texttt{not}\longrightarrow \texttt{True}$$

“Order of precedence”

The precedence of logical operators is even lower than that of comparison operators.

First `x**y` `-x` `+x` `x%y` `x/y` `x*y` `x-y` `x+y`

`x==y` `x!=y` `x>=y` `x>y` `x<=y` `x<y`

`not x` `x and y` `x or y` Last

Summary

Comparisons	`== != < > <= >=`
Numerical comparison	`5 < 7`
Alphabetical ordering	`'dig' < 'dug'`
Booleans	`True` `False`
Boolean operators	`and or not`
Order of precedence	

Exercise

Predict whether these expressions will evaluate to `True` or `False`. Then try them.

1. `'sparrow' > 'eagle'`

2. `'dog' < 'Cat' or 45 % 3 == 15`

3. `60 - 45 / 5 + 10 == 1`

Exercise

Predict the outcome in the following numerical examples and then run

```
                           Expression:
(6 <= 6) and (5 < 3)

(6 <= 6) or (5 < 3)

(5 != 6)

(5 < 3) and (6 <= 6) or (5 != 6)

(5 < 3) and ((6 <= 6) or (5 != 6))

not((5 < 3) and ((6 <= 6) or (5 != 6)))
```

“Syntactic sugar”

`a+= b`	`a = a + b`
`a-= b`	`a = a - b`
`a*= b`	`a = a * b`
`a /= b`	`a = a / b`
`a **= b`	`a = a ** b`
`a %= b`	`a = a % b`

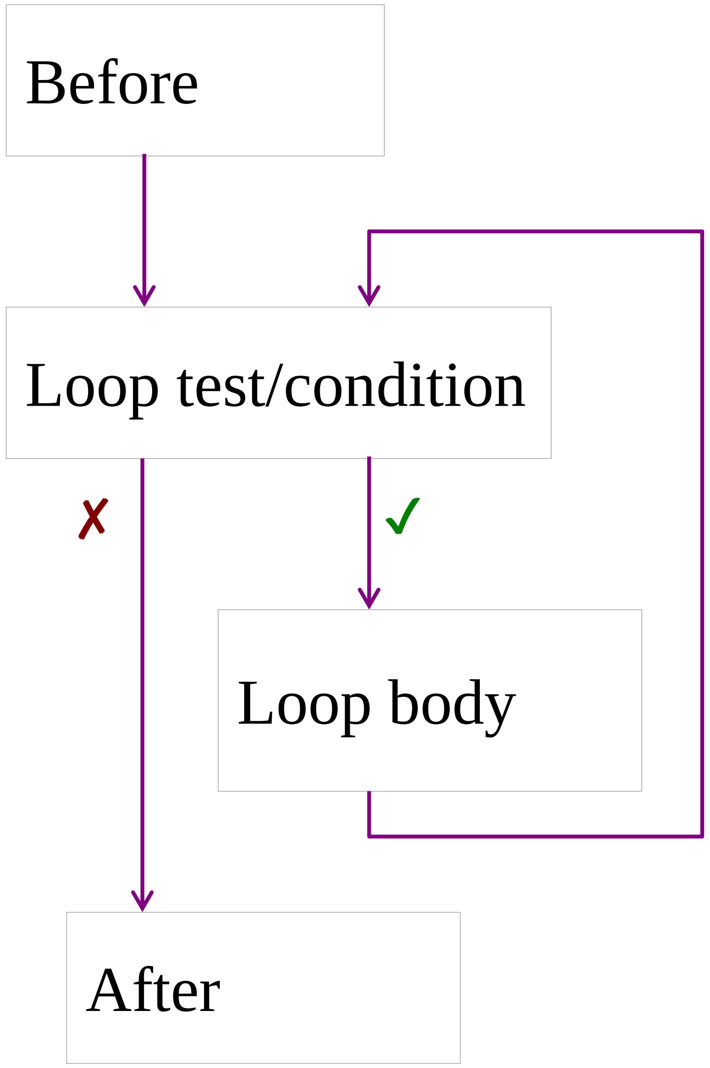
Deleting a name

```
>>> print(value)

10

>>> del value


>>> print(thing)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'thing' is not defined
```

Known variable

Unknown variable

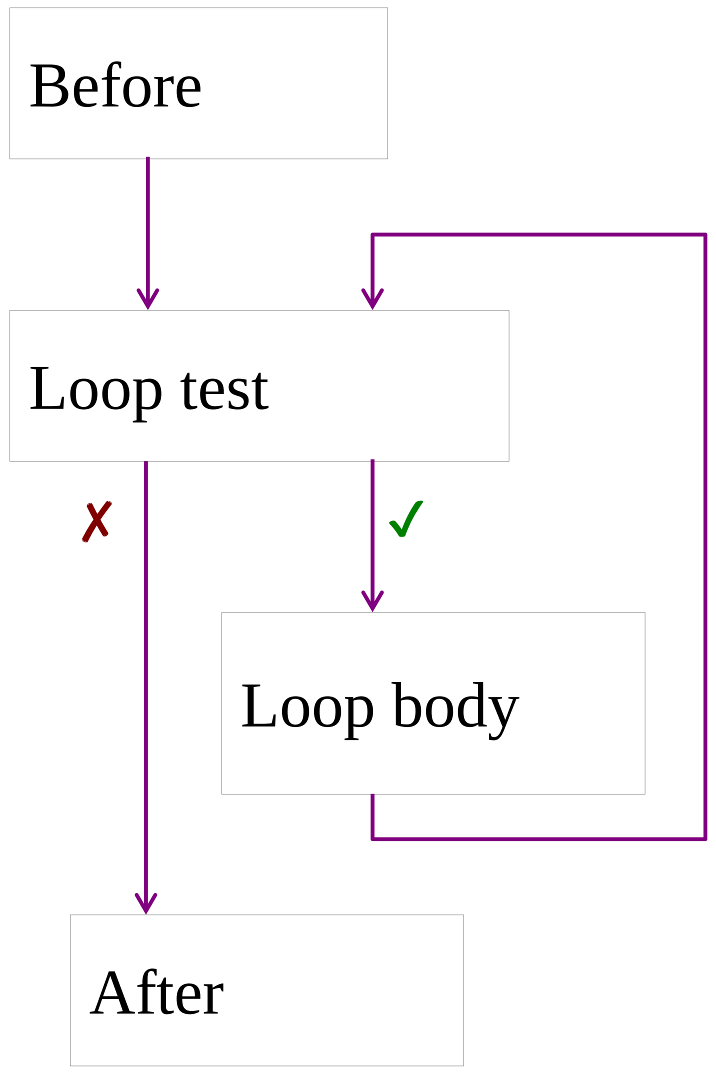
Loops – The mechanics

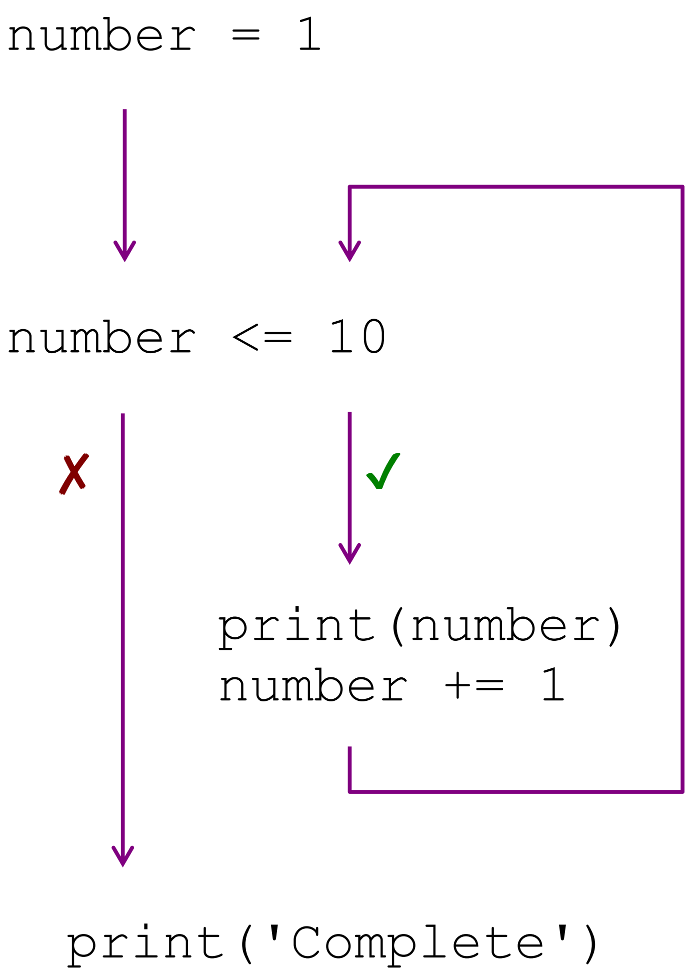


Should the loop run (again)?

What is run each loop?

Loop example: Count from 1 to 10



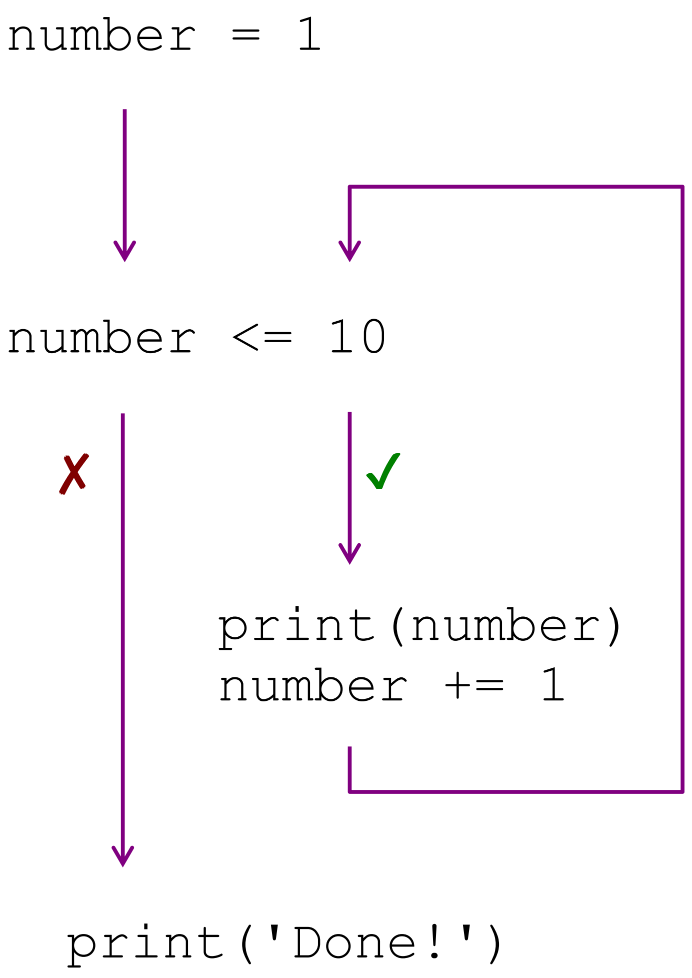


```
number = 1

number <= 10

print(number)
number += 1

print('Complete')
```

Loop example: Count from 1 to 10

```
number = 1

while number <= 10 :

␣␣␣␣print(number)
␣␣␣␣number += 1

print('Done!')
```

keyword



Loop test: Count from 1 to 10

```
number = 1

while number <= 10 :

    print(number)
    number += 1

print('Done!')
```

“while” keyword

loop test

colon

A loop becomes an infinite loop if a condition never becomes FALSE. Use caution with while loops.

Loop body: Count from 1 to 10

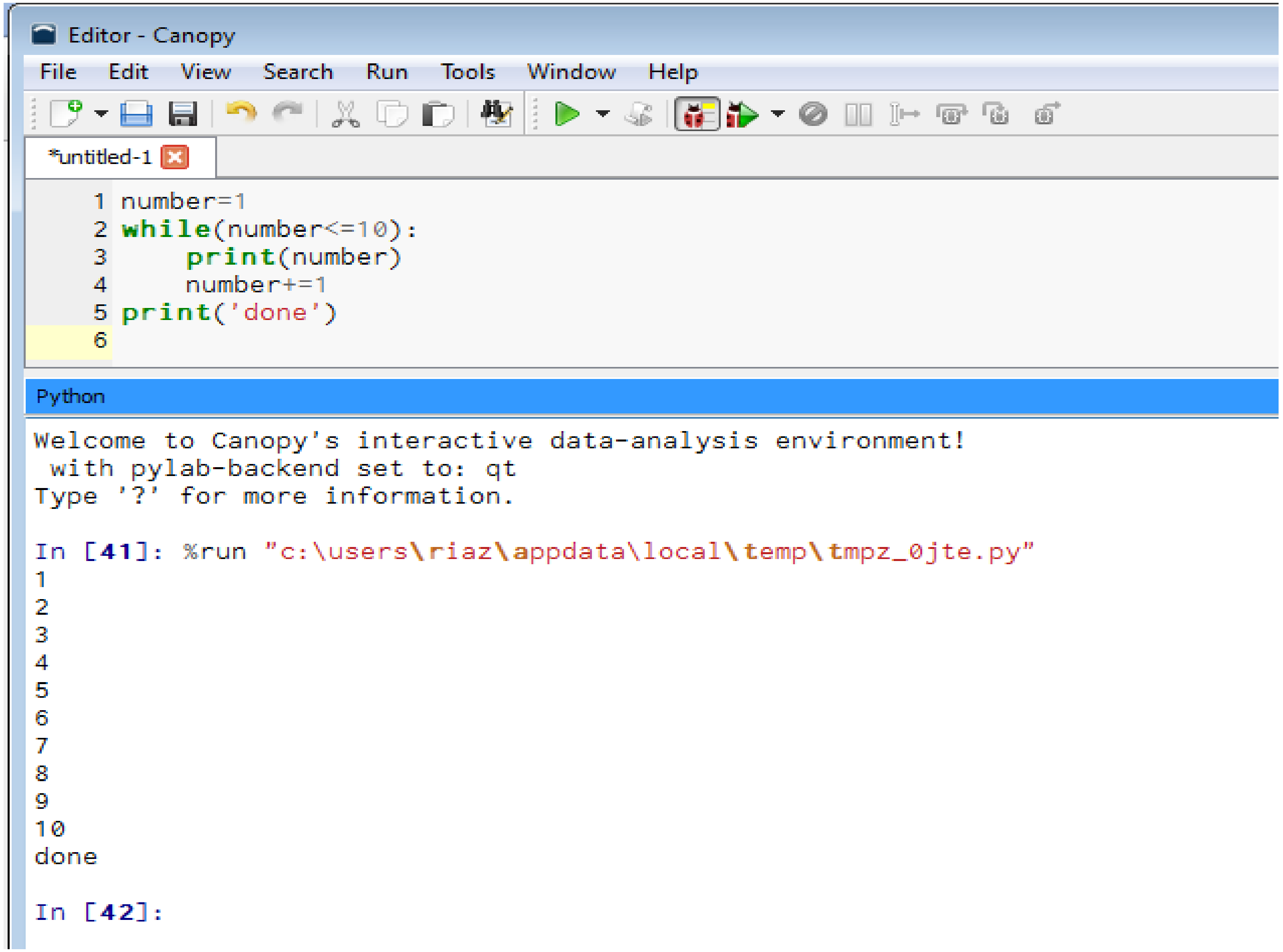
```
number = 1

while number <= 10 :

␣␣␣␣print(number)
␣␣␣␣number += 1

print('Done!')
```

loop body

Four spaces' indentation indicate a "block" of code.

The first unindented line marks the end of the block.

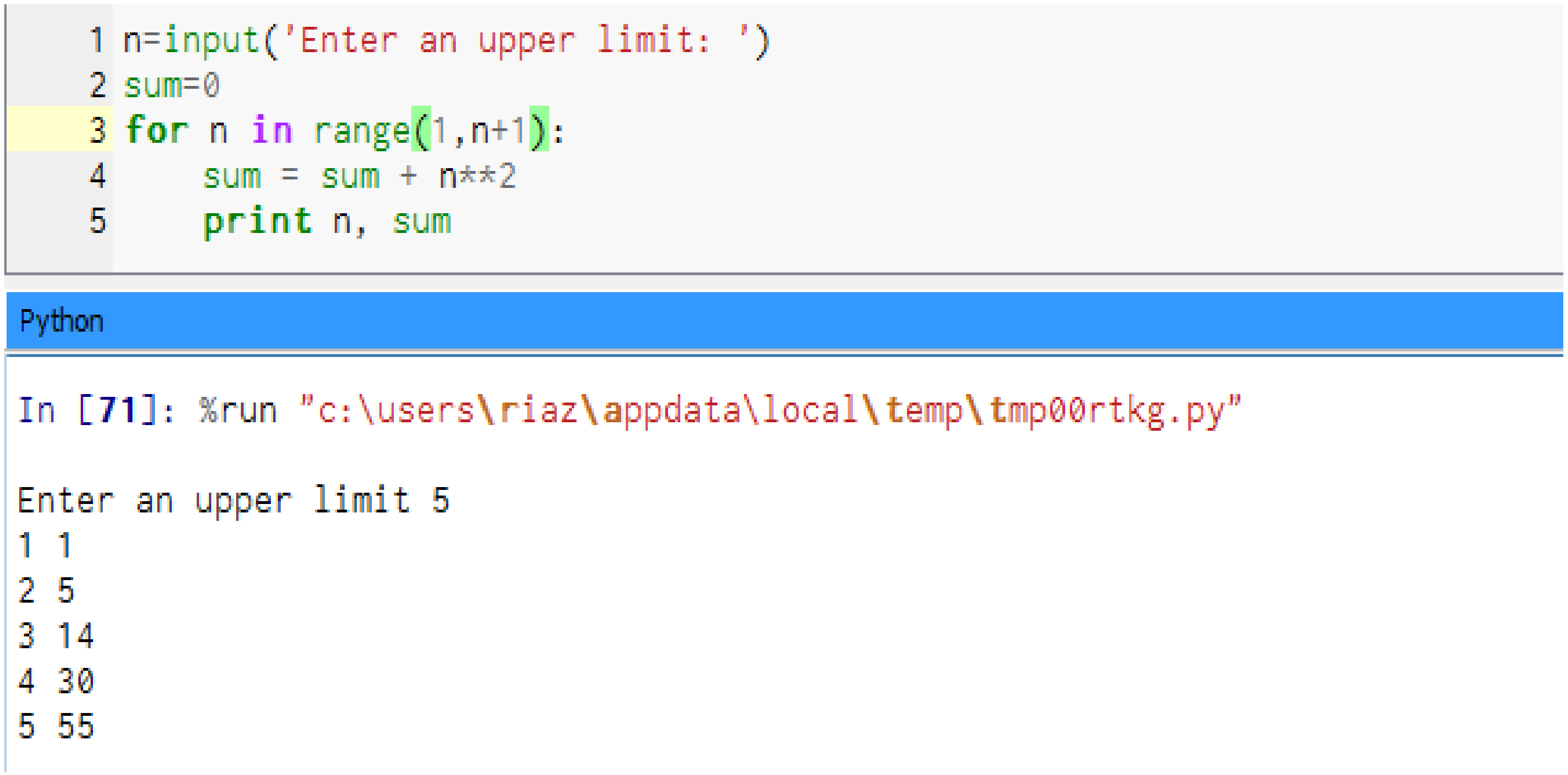
Loop example: Code and output



```
number=1
while(number<=10):
    print(number)
    number+=1
print('done')

```

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [41]: %run "c:\users\riaz\appdata\local\temp\tmpz_0jte.py"
1
2
3
4
5
6
7
8
9
10
done

In [42]:
```

For Loop for summing: code and output



```
n=input('Enter an upper limit: ')
sum=0
for n in range(1,n+1):
    sum = sum + n**2
    print n, sum
```

Python

```
In [71]: %run "c:\users\riaz\appdata\local\temp\tmp00rtkg.py"

Enter an upper limit 5
1 1
2 5
3 14
4 30
5 55
```

Keep looping while … ?

uncertainty > tolerance

```
while uncertainty > tolerance :
␣␣␣␣Do stuff.
␣␣␣␣
␣␣␣␣
␣␣␣␣
```

The “for loop” for adding

```
numbers = [45, 76, -23, 90, 15]

sum = 0

for number in numbers :

␣␣␣␣sum += number

print(sum)
```

Set up before the loop

Processing in the loop

Results after the loop

Lists

Lists are sequences of values, very similar to strings, except that each element can be of any type – they are *heterogeneous*. The syntax for creating a list is [……] where each element is separated with a ,

```
['American','Asian','Bermudan','Binary`,……]

[3.141592653589793,1.5707963267948966, 0.0]

[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

Programs usually don’t operate on single values, but on whole collections of them.

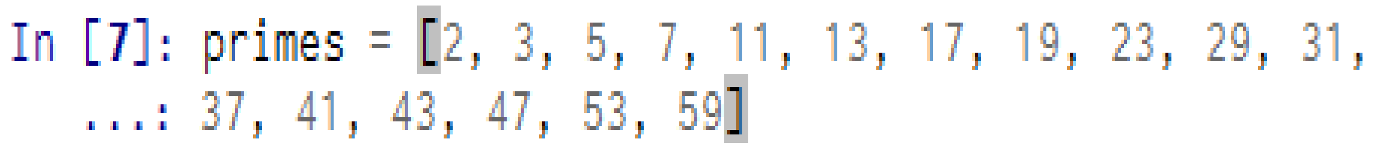
What is a list?

Consider the first list on the previous slide (more lengthier)

American, Asian, Bermudan, Binary, Cliquet, Lookback, Parisian, Passport, …, Vanilla

A *sequence* of values	The names of option's contracts
Values stored in order	Alphabetic
Individual value identified by position in the sequence	"Binary" is the name of the element number 3 in the list

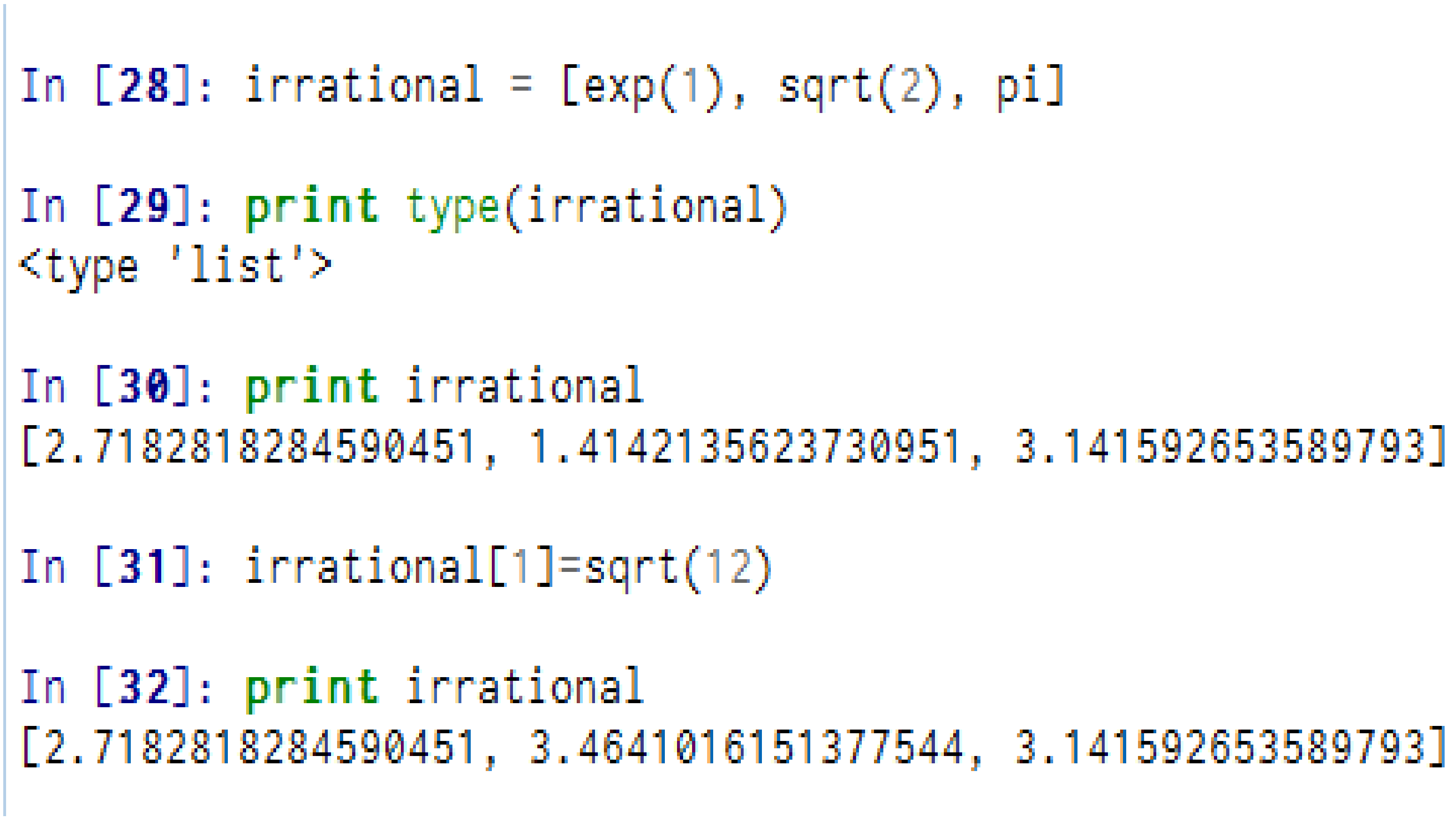
Creating a list in Python



```
In [7]: primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
   ...: 37, 41, 43, 47, 53, 59]
```

A *sequence* of values	The prime numbers less than sixty
Values stored in order	Numerical order
Individual value identified by position in the sequence	17 is the element number six

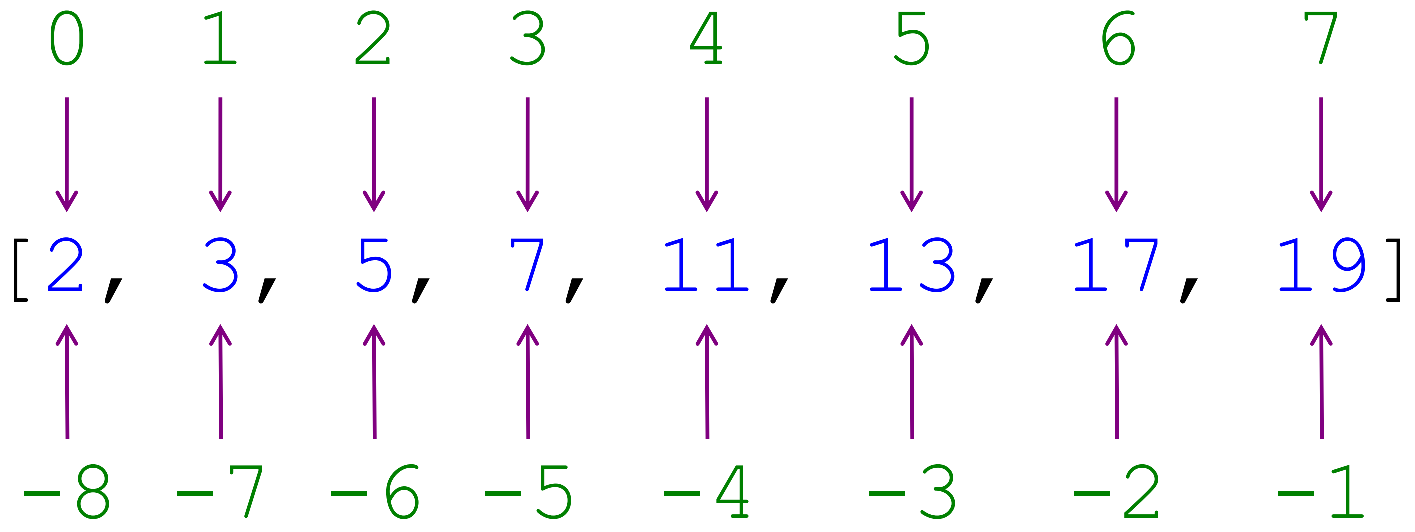
A list of irrationals



```
In [28]: irrational = [exp(1), sqrt(2), pi]

In [29]: print type(irrational)
<type 'list'>

In [30]: print irrational
[2.718281828459045l, 1.4142135623730951, 3.141592653589793]

In [31]: irrational[1]=sqrt(12)

In [32]: print irrational
[2.718281828459045l, 3.4641016151377544, 3.141592653589793]
```

Counting from the end – indexing from the back

```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

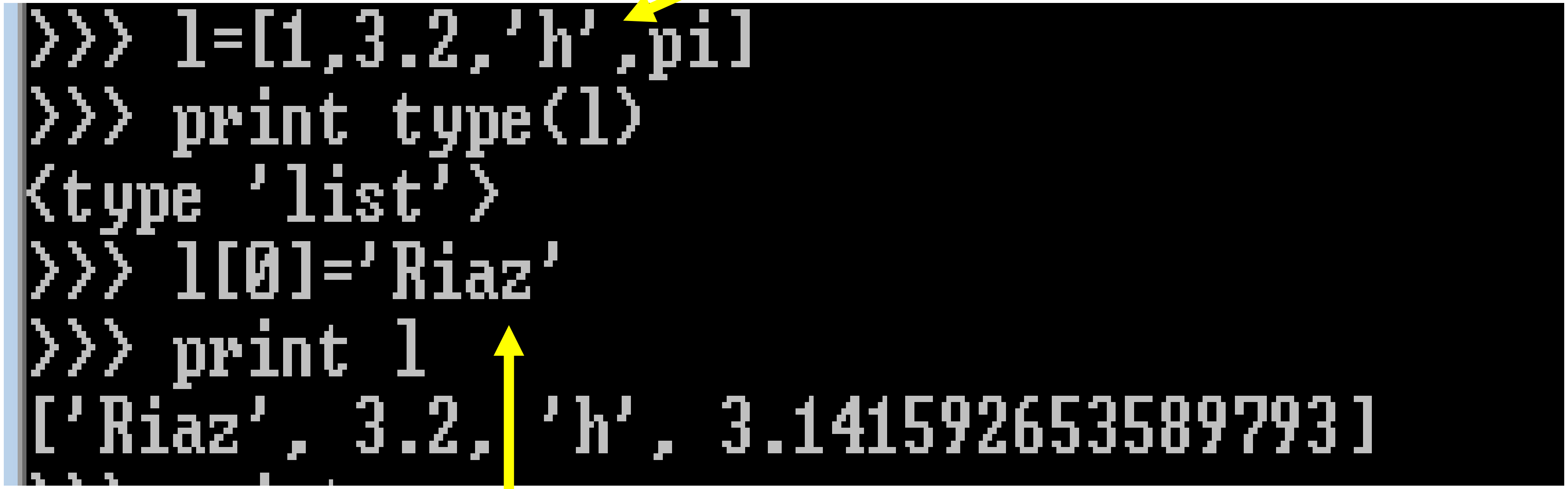


getting at the last item

```
>>> primes[-1]
19
```

Elements in a list can differ in <type>

4 elements of type int, float, character, float in turn



```
>>> l=[1,3.2,'h',pi]
>>> print type(l)
<type 'list'>
>>> l[0]='Riaz'
>>> print l
['Riaz', 3.2, 'h', 3.141592653589793]
```

First element in list changed to a string

Inserting element using list methods

List methods are an alternative, more readable way of inserting elements.

`append()` adds an element to the end of a list:

```
In [1]: b = [0.1, 0.2]
In [2]: b.append(0.9)
In [3]: b
Out[3]: [0.1, 0.2, 0.9]
```

`extend()` appends all elements of another list:

```
In [4]: b.extend([7, 8])
In [5]: b
Out[5]: [0.1, 0.2, 0.9, 7, 8]
```

`insert(i, x)` inserts x before ith element:

```
In [6]: b.insert(1, 5)
In [7]: b
Out[7]: [0.1, 5, 0.2, 0.9, 7, 8]
```

Further insertion using list methods

List methods are an alternative, more readable way of inserting elements.

`pop(i)` removes and returns the i^{th} element:

```
In [1]: b = [0.1,0.2,0.3]
In [2]: b.pop(1) # removes 0.2 from list and returns value
Out [2]: 0.2
In[3]: b
Out[3]:[0.1, 0.3]
```

If `pop()` is called with no arguments, it removes the last element of the list.

```
In [4]: b = [0.1,0.2,0.3]
In [5]: b.pop()
Out[5]: 0.3
In [6]: b
Out[6]: [0.1,0.2]
```

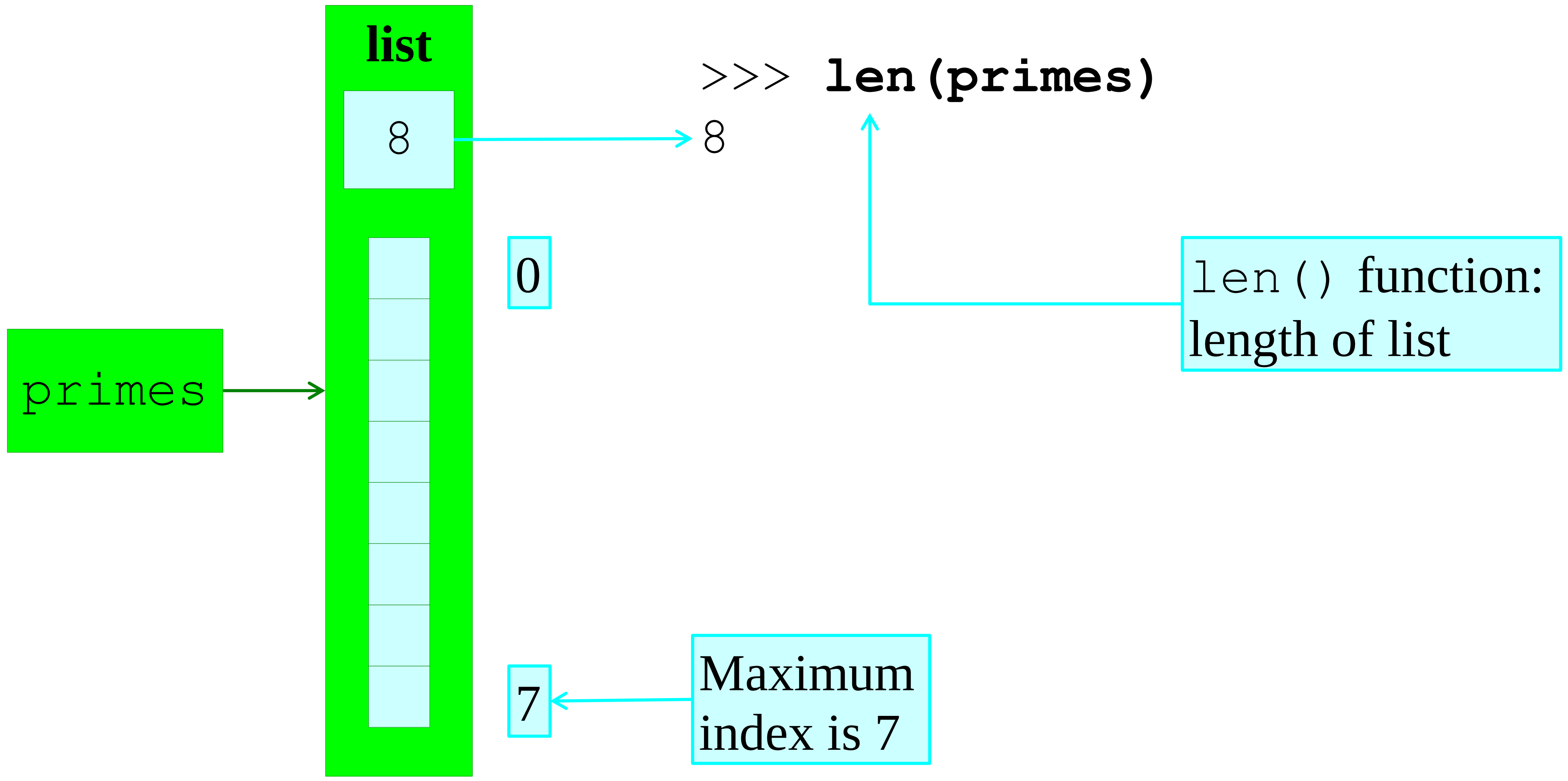
```
In [4]: del b[0]
In [5]: b
Out[5]: [0.3]
```

Deletion from a list

The **`del()`** statement can be used to remove an element.
Using `b = [0.1,0.2,0.3]`
from the previous slide :

```
In [7]: del b[1]
In [8]: b
Out[8]: [0.1, 0.3]
```

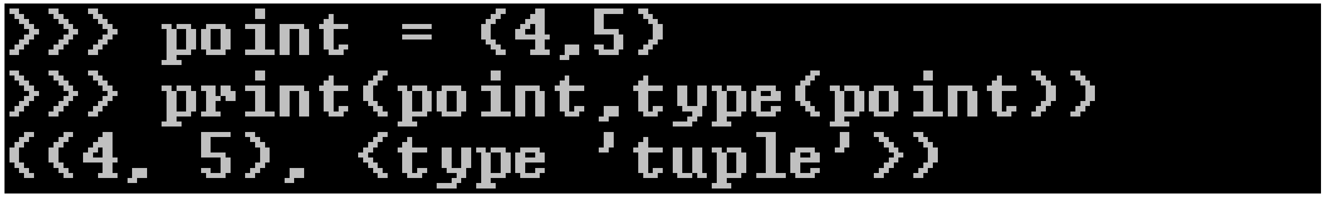
Length of a list – use len



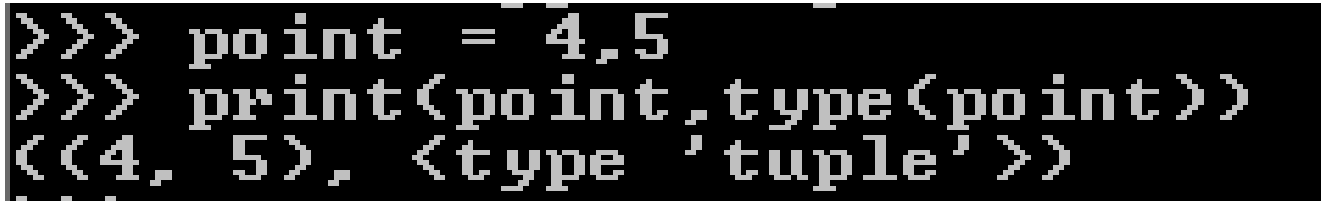
```
>>> len(primes)
8
```

Tuples

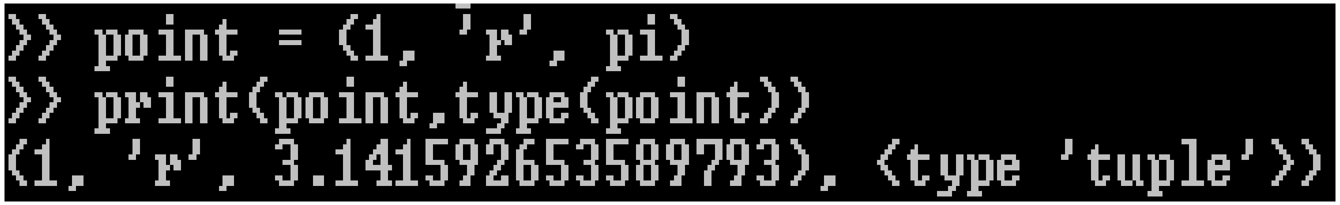
Tuples are like lists, except that they cannot be modified once created, that is they are *immutable*. In Python, lists, are easily created using the syntax （...， ...， ...）, or even ...， ...：



```
>>> point = (4,5)
>>> print(point,type(point))
((4, 5), <type 'tuple'>)
```



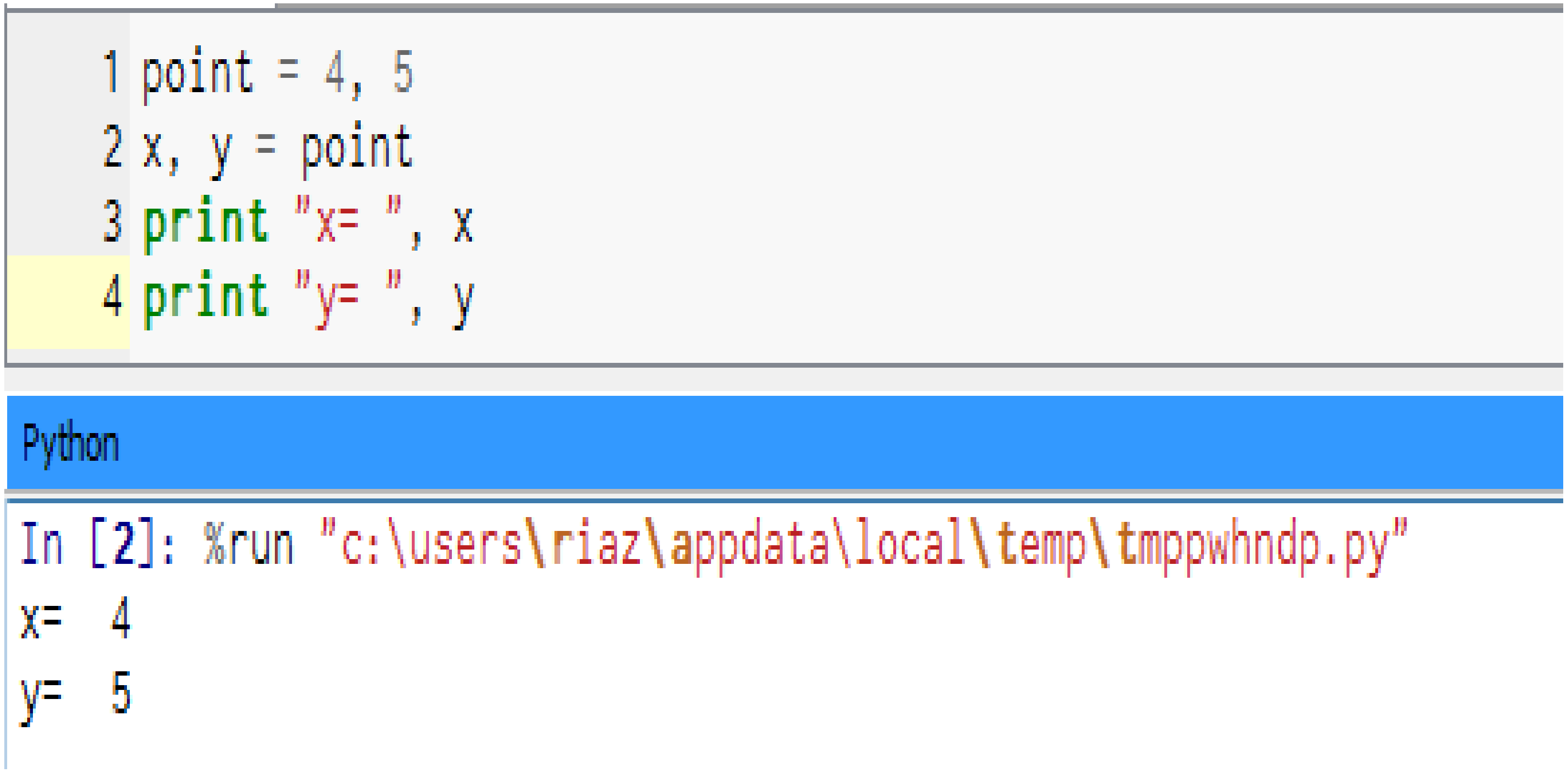
```
>>> point = 4,5
>>> print(point,type(point))
((4, 5), <type 'tuple'>)
```



```
>> point = (1, 'r', pi)
>> print(point,type(point))
(1, 'r', 3.141592653589793), <type 'tuple'>)
```

Unpacking tuples

Tuples can be unpacked by assigning it to a comma-separated list of variables



```
point = 4, 5
x, y = point
print "x= ", x
print "y= ", y
```

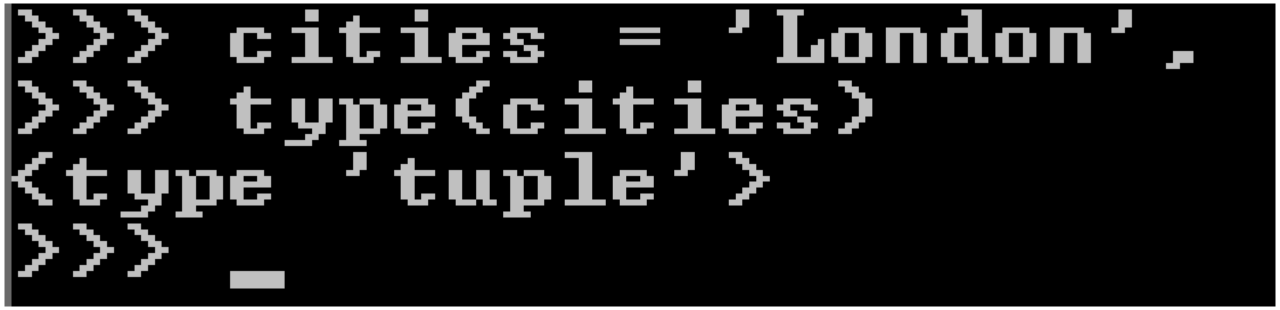
Python

```
In [2]: %run "c:\users\riaz\appdata\local\temp\tmppwhndp.py"
x=  4
y=  5
```

Trying to assign a new value to an element in a tuple results in an error.

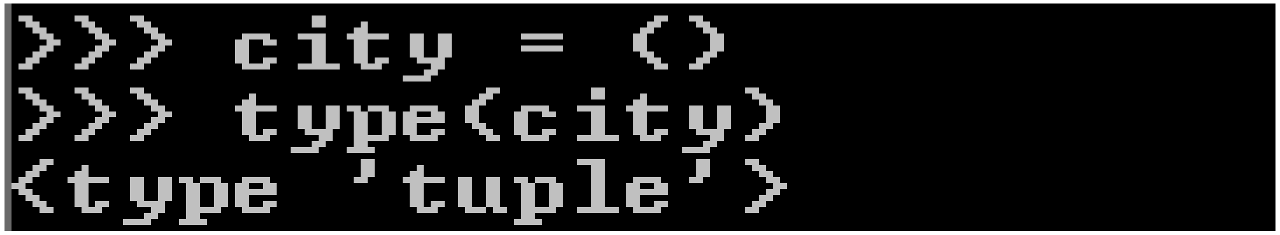
Simpler Tuples

To construct a **single-element** tuple, put an extra comma:



```
>>> cities = 'London',
>>> type(cities)
<type 'tuple'>
>>> _
```

An empty tuple is denoted by ()



```
>>> city = ()
>>> type(city)
<type 'tuple'>
```

Lists to Tuples

To **convert** a list to a tuple, use the function **tuple**():

```
In [3]: stuff = [7, ’xyz’]

In [4]: tuple(stuff)

Out[4]: (7, ’xyz’)

In [5]: stuffs=tuple(stuff)

In [5]: print stuffs

Out [5]: (7, ’xyz’)
```

Indexing and slicing

Indexing and slicing works as for lists:

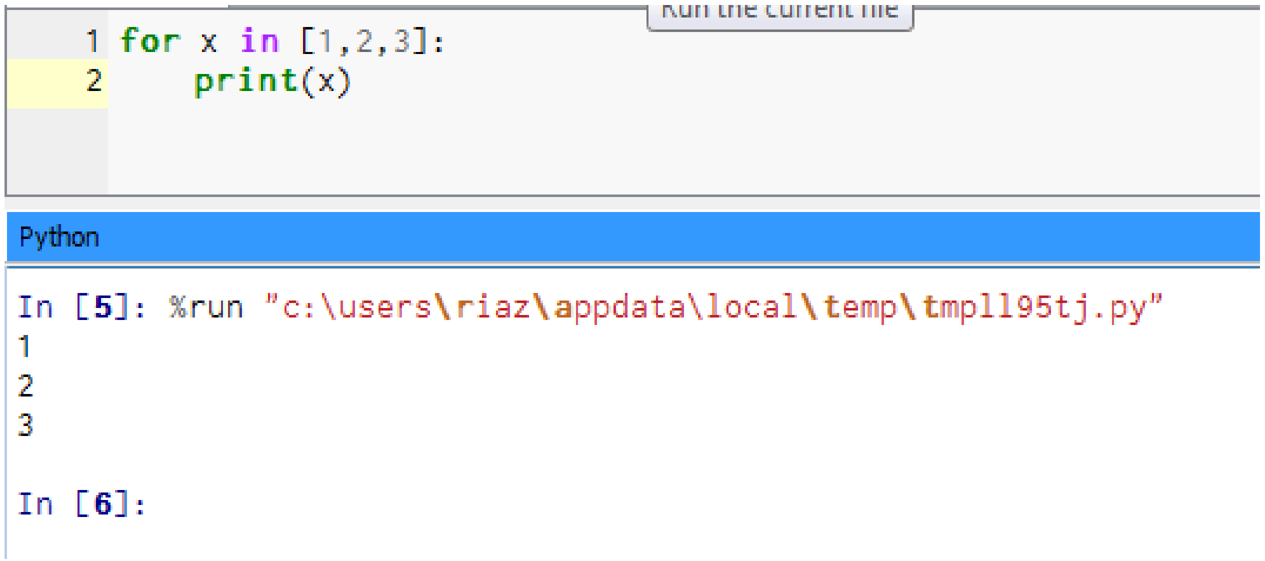
```
In [1]: address = ’UK’, ’London’, ’WC1E
6BT’

In [2]: address[1]

Out[2]: ’London’

In [3]: address[1][0:3]

Out[3]: ’Lon’
```

However assignment is not allowed. So e.g. the following is not allowed:

```
In [4]: address[2] = ‘NW1 1AB’
```

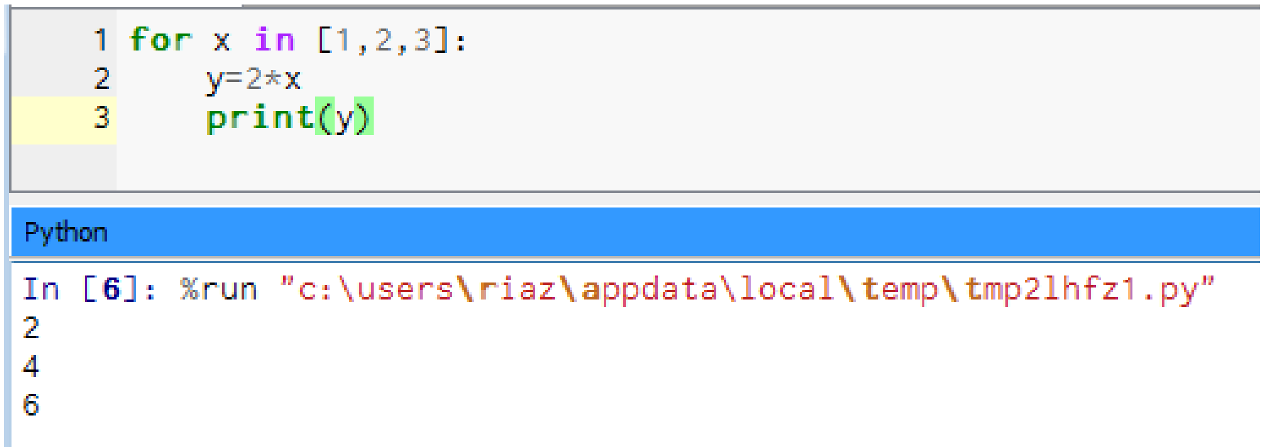
Loops again!

In Python, loops can be programmed in a number of different ways. The most common is the `for` loop, which is used together with iterable objects, such as lists. The basic syntax is:



```
for x in [1,2,3]:
    print(x)
```

```
In [5]: %run "c:\users\riaz\appdata\local\temp\tmpl195tj.py"
1
2
3

In [6]:
```



```
for x in [1,2,3]:
    y=2*x
    print(y)
```

```
In [6]: %run "c:\users\riaz\appdata\local\temp\tmp2lhfz1.py"
2
4
6
```

The `range()` function

It is tedious to write:

```
In [1]: a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is shorter and equivalent:

```
In [2]: a = range(10)
```

`range(n)` returns the list of integers from 0 up to but not including n.
`range(m, n)` returns the list of integers from **m** up to but not including **n**:

```
In [3]: range(5, 10)
```

Out[3]: [5, 6, 7, 8, 9]

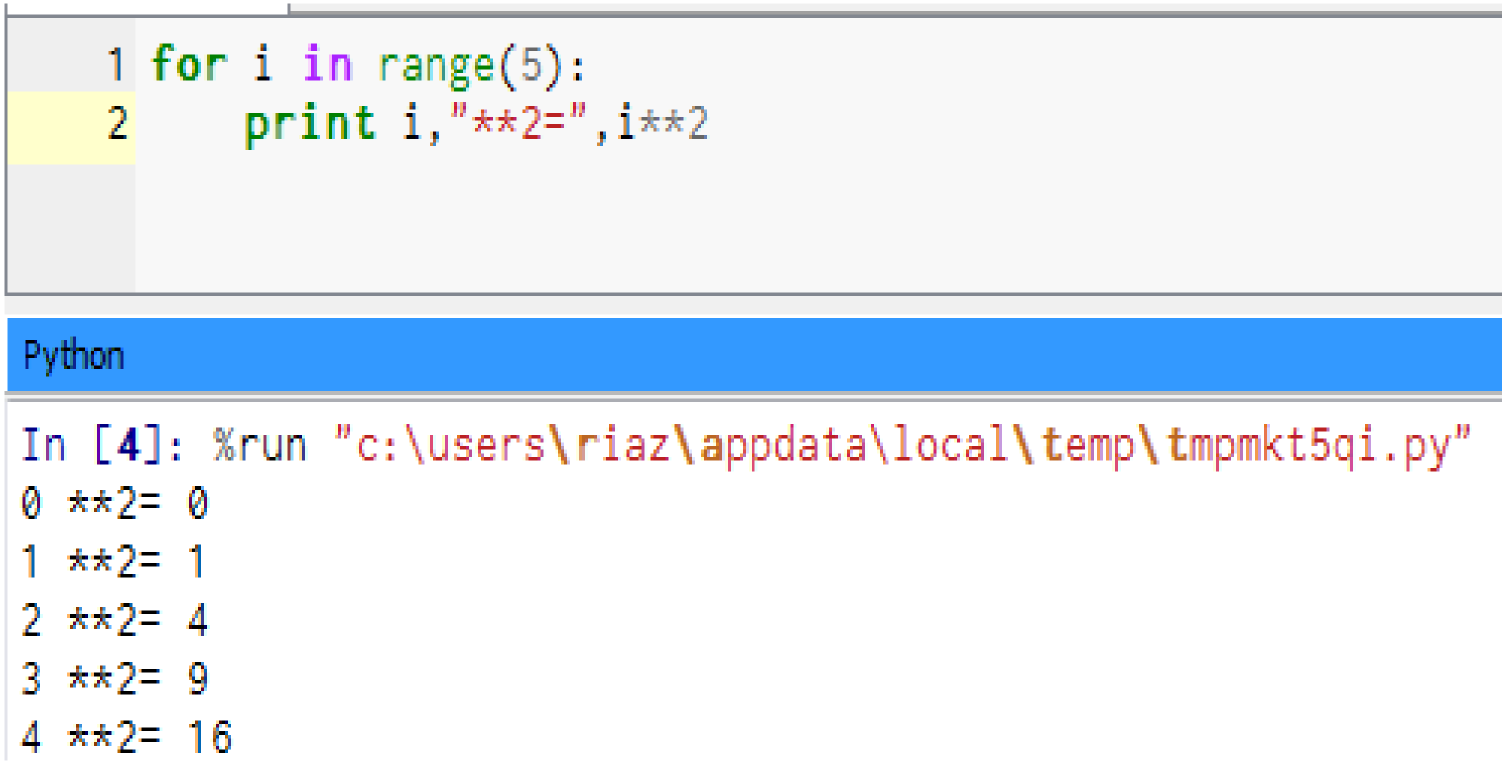
range(m, n, s) returns the list containing every s^{th} **integer** from **m** up to but not including **n**:

In [4]: **range**(4, 10, 2)

Out[4]: [4, 6, 8]

Iteration over integers I

A common use case of range() is iteration over lists of integers.



```python
for i in range(5):
    print i,"**2=",i**2
```

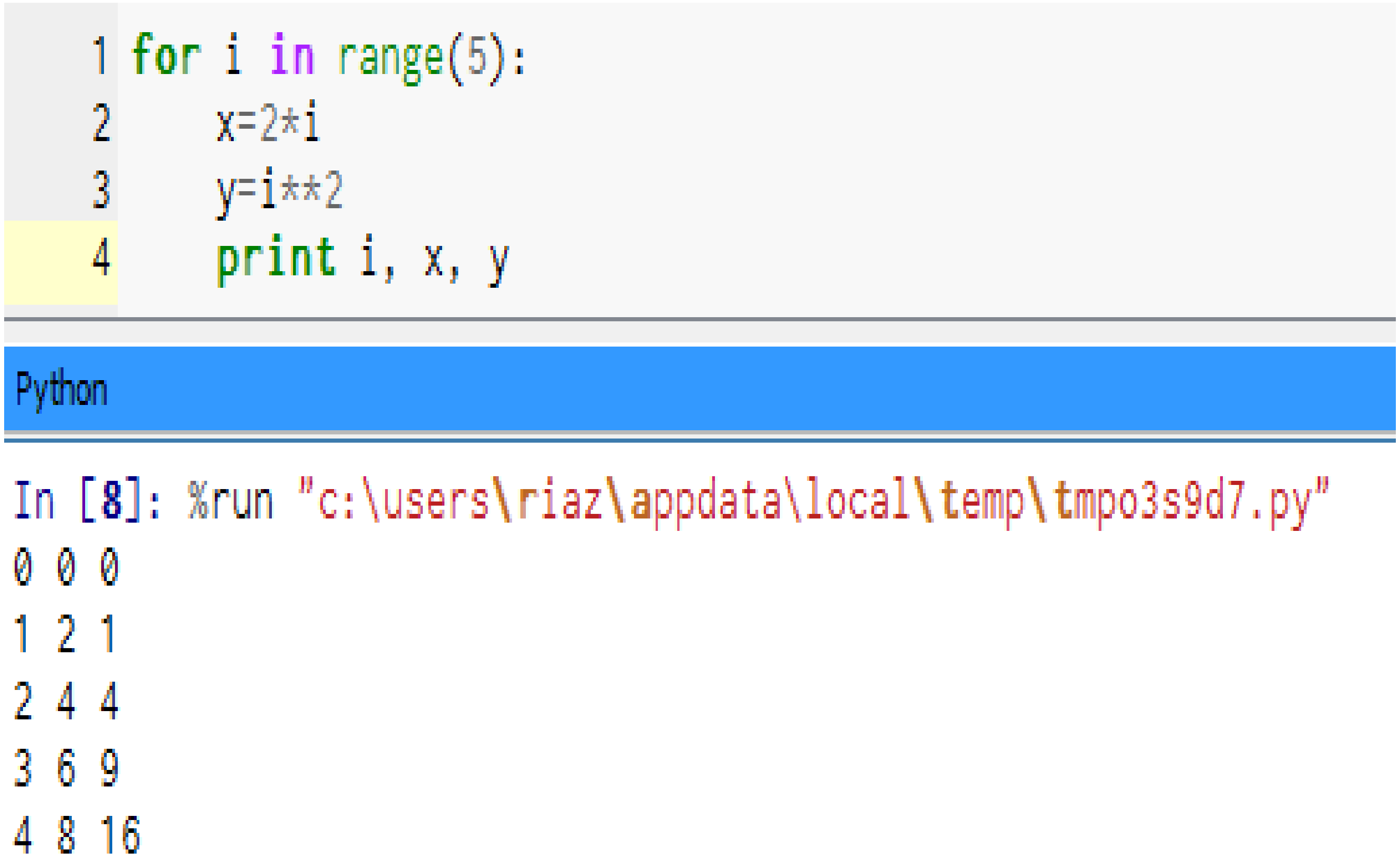
Python

```
In [4]: %run "c:\users\riaz\appdata\local\temp\tmpmkt5qi.py"
0 **2= 0
1 **2= 1
2 **2= 4
3 **2= 9
4 **2= 16
```

range([start,] stop[, step]) -> list of integers

print range(1,10,2) prints out [1, 3, 5, 7, 9]

Iteration over integers II



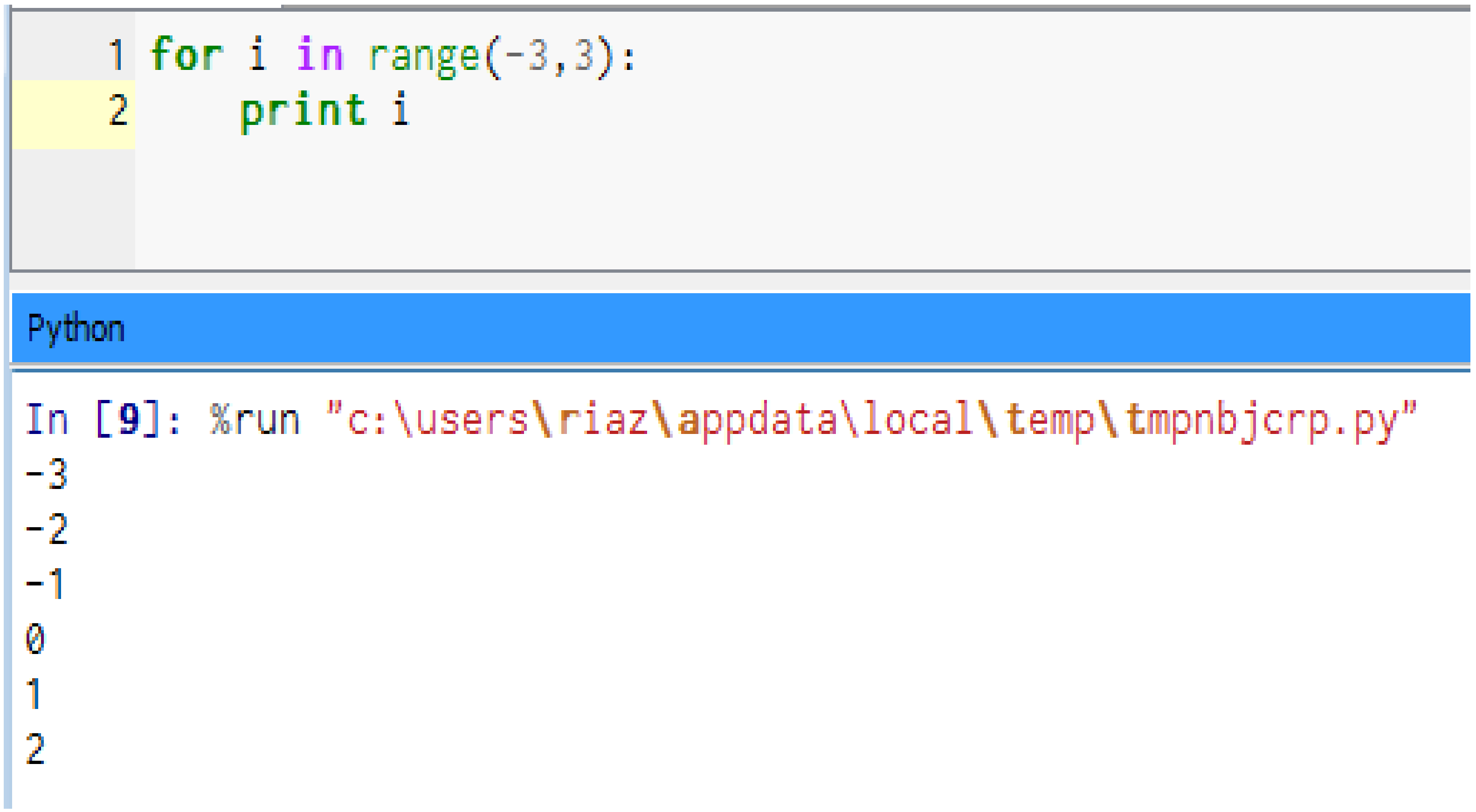
```
for i in range(5):
    x=2*i
    y=i**2
    print i, x, y
```

Python

```
In [8]: %run "c:\users\riaz\appdata\local\temp\tmpo3s9d7.py"
0 0 0
1 2 1
2 4 4
3 6 9
4 8 16
```

Note `range(5)` does not include 5

Iteration over integers III



```
for i in range(-3,3):
    print i
```

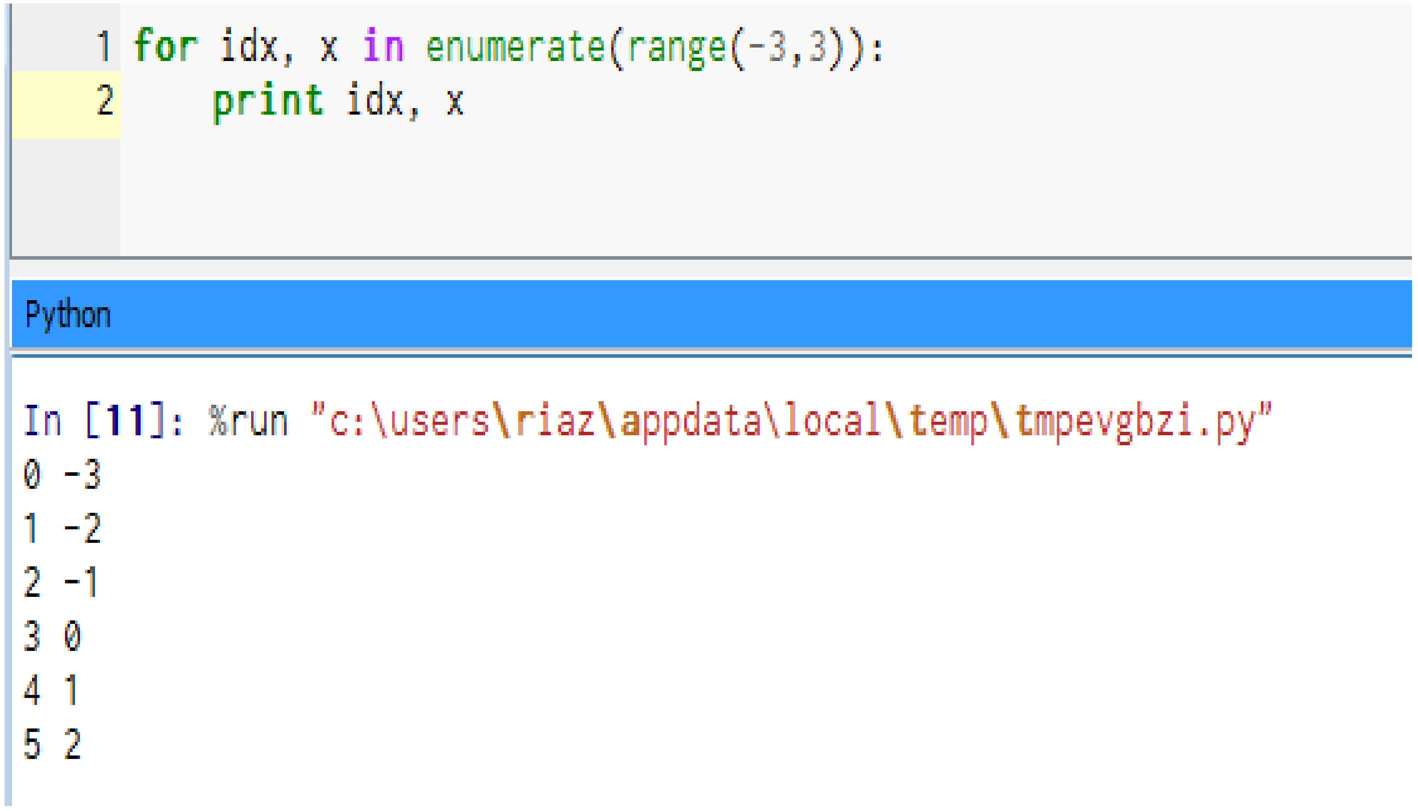
Python

```
In [9]: %run "c:\users\riaz\appdata\local\temp\tmpnbjcrp.py"
-3
-2
-1
0
1
2
```

Again note `range(3)` does not include 3

Iteration over integers IV

Sometimes it is useful to have access to the indices of the values when iterating over a list. We can use the enumerate function for this:



```
for idx, x in enumerate(range(-3,3)):
    print idx, x
```

```
In [11]: %run "c:\users\riaz\appdata\local\temp\tmpevgbzi.py"
0 -3
1 -2
2 -1
3 0
4 1
5 2
```

Exercise

Track what is happening to this list at each stage. Do this initially by hand. After each line, work out what you think the numbers will be and then check by printing out.

```
>>> numbers = [5, 7, 11, 13, 17, 19, 29, 31]

>>> numbers[1] = 3

>>> del numbers[3]

>>> numbers[3] = 37

>>> numbers[4] = numbers[5]

>>> numbers = [5, 7, 11, 13, 17, 19, 29, 31]
```

Using the `append()` method

```
>>> print(primes)
[2, 3, 5, 7, 11, 13, 17, 19]

>>> primes.append(23)

>>> primes.append(29)

>>> primes.append(31)

>>> primes.append(37)

>>> print(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

The function doesn’t return any value.

It modifies the list itself.

Other methods on lists: `reverse()`

```
>>> numbers = [4, 7, 5, 1]

>>> numbers.reverse()

>>> print(numbers)
[1, 5, 7, 4]
```

The function doesn’t return any value.

It modifies the list itself.

Other methods on lists: `sort()`

```
>>> numbers = [4, 7, 5, 1]

>>> numbers.sort()

>>> print(numbers)
[1, 4, 5, 7]
```

The function does not return the sorted list.

It sorts the list itself.

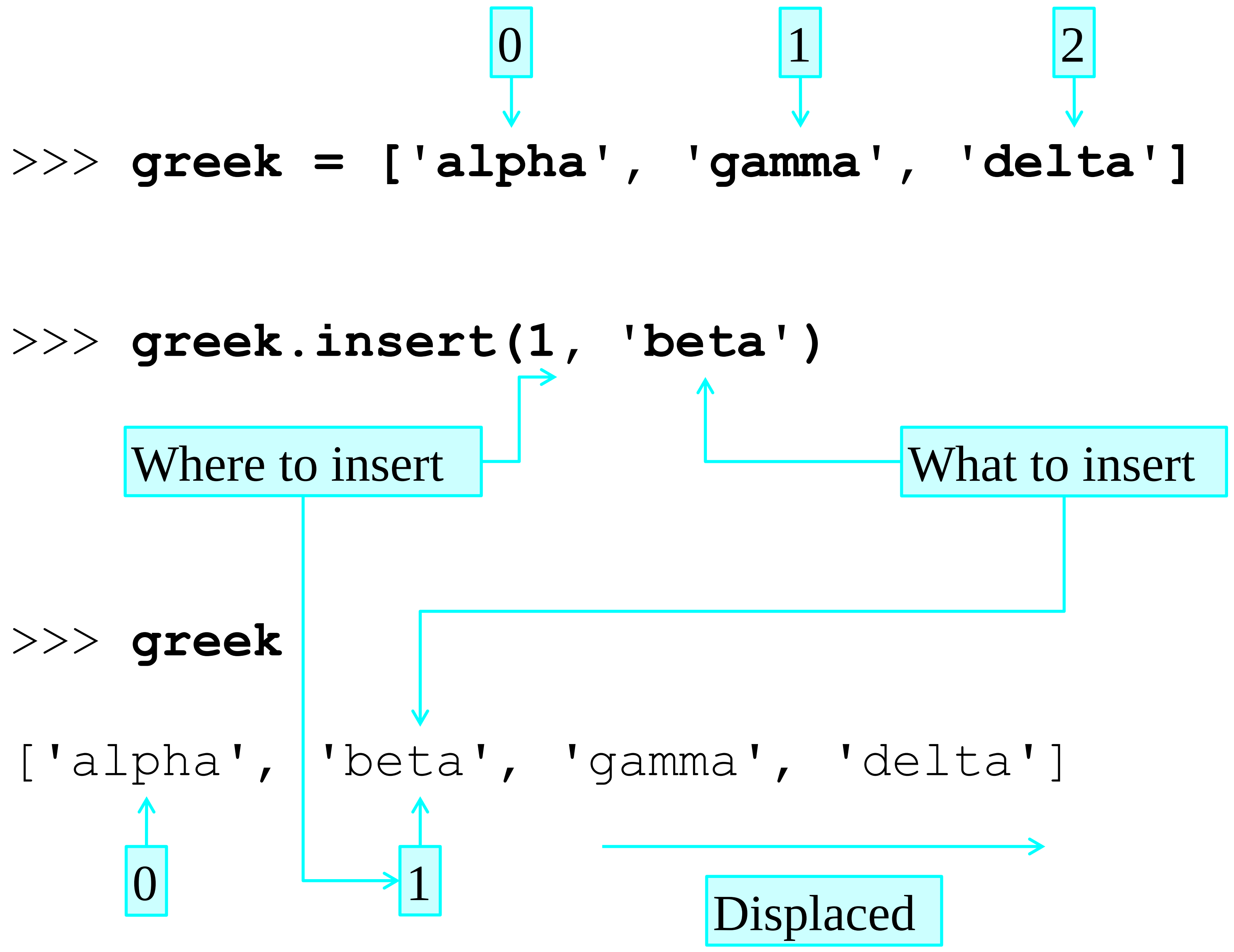
Numerical order.

Other methods on lists: `sort()`

```
>>> greek = ['alpha', 'delta', 'beta', 'gamma']

>>> greek.sort()

>>> print(greek)
['alpha', 'beta', 'delta', 'gamma']
```

Alphabetical order
of the *words*.

Other methods on lists: `insert()`



```
>>> greek = ['alpha', 'gamma', 'delta']

>>> greek.insert(1, 'beta')

>>> greek

['alpha', 'beta', 'gamma', 'delta']
```

Other methods on lists: `remove()`

```
>>> numbers = [7, 4, 8, 7, 2, 5, 4]

>>> numbers.remove(8)

>>> print(numbers)
[7, 4, 7, 2, 5, 4]
```

Value to remove

c.f. `del numbers[2]`

Index to remove

Other methods on lists: `remove()`

```
>>> print(numbers)
[7, 4, 7, 2, 5, 4]
```

There are two instances of 4.

```
>>> numbers.remove(4)

>>> print(numbers)
[7, 7, 2, 5, 4]
```

Only the first instance is removed

Adding to a list : “+”

```
>>> primes
[2, 3, 5, 7, 11, 13, 17, 19]
```

Concatenation operator

List to add

```
>>> primes + [23, 29, 31]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Concatenation

Create a new list

```
>>> newlist = primes + [23, 29, 31]
```

Update the list

```
>>> primes  = primes + [23, 29, 31]
```

Augmented assignment

```
>>> primes += [23, 29, 31]
```

Is an item in a list? — 1

```
>>> odds = [3, 5, 7, 9]

>>> odds.remove(2)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Does not include 2

Try to remove 2

Hard error

x must be in the list before it can be removed

✗

Is an item in a list? — 2

```
>>> odds = [3, 5, 7, 9]

>>> 2 in odds

False

>>> 3 in odds

True

>>> 2 not in odds

True
```

Precedence

First

```
x**y   -x     +x     x%y    x/y    x*y    x-y    x+y

x==y   x!=y   x>=y   x>y    x<=y   x<y

x not in y           x in y

not x    x and y     x or y
```

Last

The list now contains every operator we meet in this course.

Ranges of numbers again

via `list()`

```
range(10)        → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Start at 0

```
range(3, 10)     → [3, 4, 5, 6, 7, 8, 9]
```

```
range(3, 10, 2)  → [3, 5, 7, 9]
```

Every *n*th number

```
range(10, 3, -2) → [10, 8, 6, 4]
```

Negative steps

Indices of lists

```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]

>>> len(primes)

8

>>> list(range(8))

[0, 1, 2, 3, 4, 5, 6, 7]
```

valid indices

$$\begin{tikzcd}
0 \arrow[d] & 1 \arrow[d] & 2 \arrow[d] & 3 \arrow[d] & 4 \arrow[d] & 5 \arrow[d] & 6 \arrow[d] & 7 \arrow[d] \\
{[2,} & {3,} & {5,} & {7,} & {11,} & {13,} & {17,} & {19]}
\end{tikzcd}$$

Tuples as single objects — 1

```
>>> x = 20
>>> type(x)
<class 'int'>

>>> y = 3.14
>>> type(y)
<class 'float'>

>>> z = (20, 3.14)
>>> type(z)
<class 'tuple'>
```

One name → *Pair* of values

A single object

Tuples as single objects — 2

```
>>> z = (20, 3.14)

>>> print(z)

(20, 3.14)

>>> w = z

>>> print(w)

(20, 3.14)
```

Single name → Single tuple

Splitting up a tuple

```
>>> print(z)
(20, 3.14)
>>> (a,b) = z
>>> print(a)
20
>>> print(b)
3.14
```

Two names → Single tuple

How tuples are like lists

```
>>> z = (20, 3.14)
>>> z[1]
3.14
>>> len(z)
2
>>> z + (10, 2.17)
(20, 3.14, 10, 2.17)
```

Indices

Length

Concatenation

How tuples are *not* like lists

```
>>> z = (20, 3.14)

>>> z[0] = 10

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
           support item assignment
```

“Immutable”

Additional downloadable modules

Numerical

`numpy`

`scipy`

Graphics

`matplotlib`

Databases

`pyodbc`	
`psycopg2`	PostgreSQL
`MySQLdb`	MySQL
`cx_oracle`	Oracle
`ibm_db`	DB2
`pymssql`	SQL Server

Finding modules

Python:	Built-in modules
SciPy:	**Sci**entific **Py**thon modules
PyPI:	**Py**thon **P**ackage **I**ndex
Search:	“Python3 module for X”

Help with modules

```
>>> import math

>>> help(math)

NAME
    math

DESCRIPTION
    This module is always available.  It provides
    access to the mathematical functions defined
    by the C standard.


…
```

Help with module functions

```
…

FUNCTIONS

    acos(x)
        Return the arc cosine (measured in
        radians) of x.
…
```

```
>>> math.acos(1.0)

0.0
```

Help with module constants

```
…

DATA
    e = 2.718281828459045
    pi = 3.141592653589793

…
```

```
>>> math.pi

3.141592653589793
```

Functions

$$y = f(x)$$

Functions we have met and will meet

`input(`*`prompt`*`)`	`bool(`*`thing`*`)`
`len(`*`thing`*`)`	`float(`*`thing`*`)`
`open(`*`filename`*`, `*`mode`*`)`	`int(`*`thing`*`)`
`print(`*`line`*`)`	`iter(`*`list`*`)`
`type(`*`thing`*`)`	`list(`*`thing`*`)`
`ord(`*`char`*`)`	`range(`*`from`*`, `*`to`*`, `*`stride`*`)`
`chr(`*`number`*`)`	`str(`*`thing`*`)`

Not that many!

“The Python Way”:
If it is appropriate to an object,
make it a method of that object.

Why write our own functions?

Easier to …

… read

… write

… test

… fix

… improve

… add to

… develop

“Structured programming”

Defining a function

$$(y_1, y_2, y_3) = f(x_1, x_2, x_3, x_4, x_5)$$

Identify the inputs

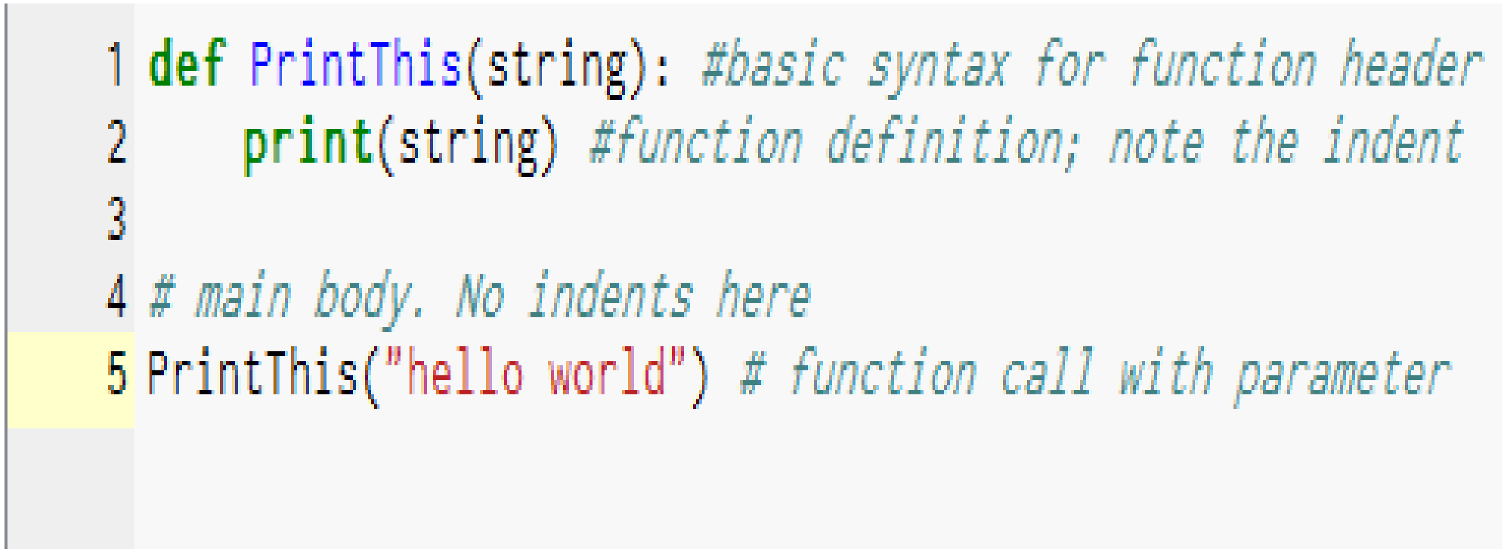
Identify the processing

Identify the outputs

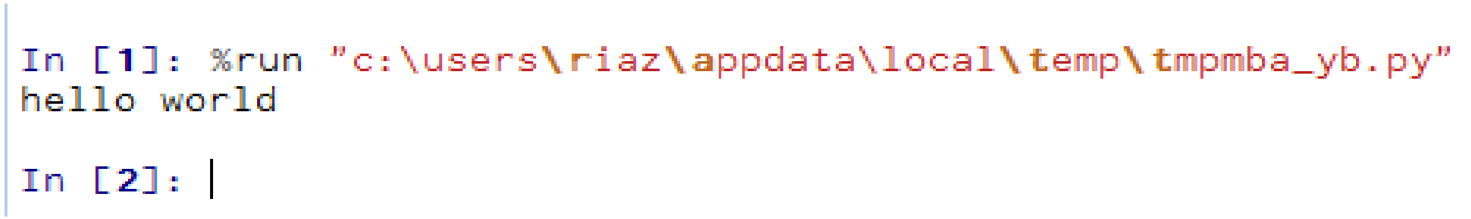
User defined functions

We now write our own functions. A function in Python is defined using the keyword def, followed by a function name, a signature within parentheses (), and a colon :

The following code, with one additional level of indentation, is the function body.

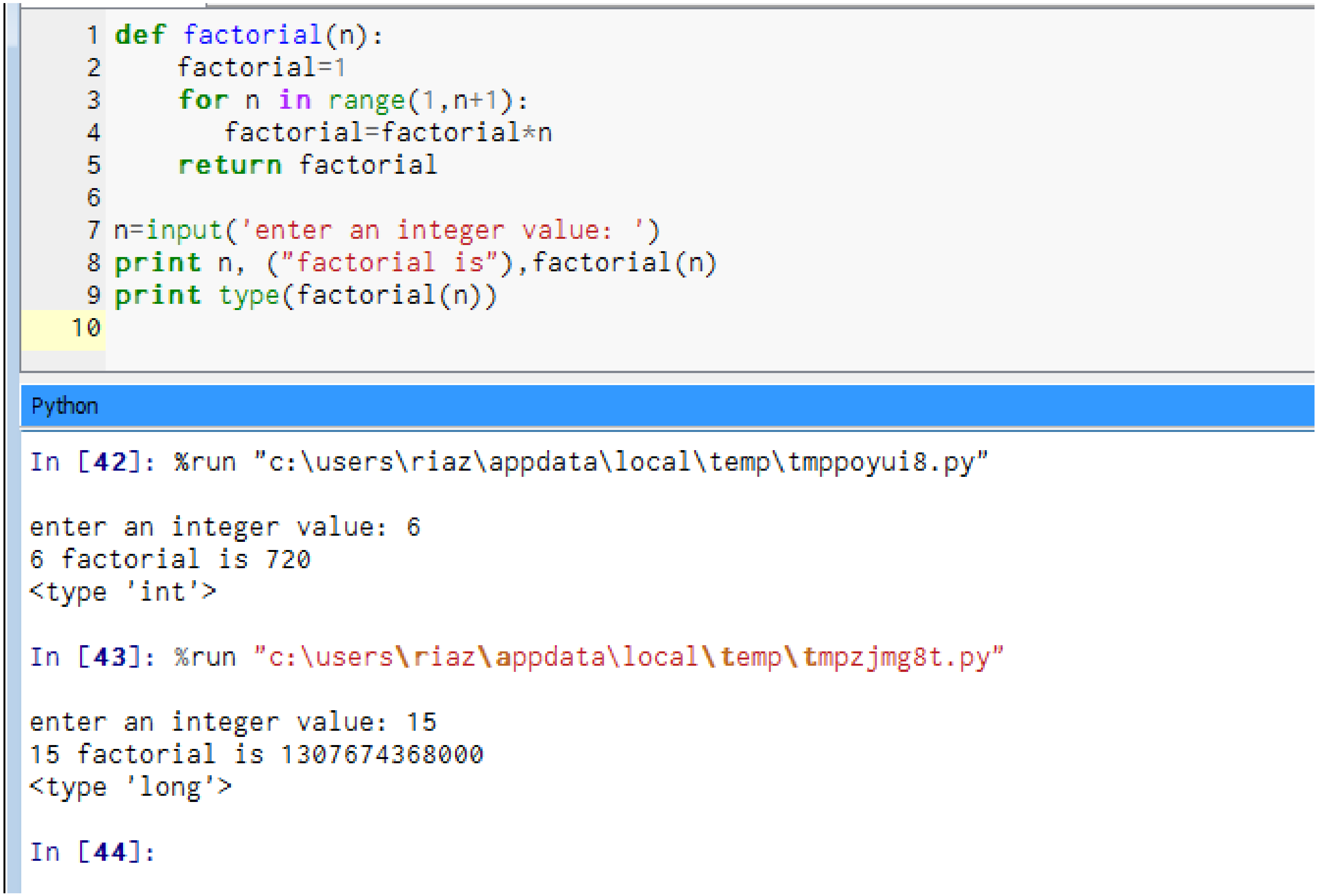


```python
def PrintThis(string): #basic syntax for function header
    print(string) #function definition; note the indent

# main body. No indents here
PrintThis("hello world") # function call with parameter
```



```
In [1]: %run "c:\users\riaz\appdata\local\temp\tmpmba_yb.py"
hello world

In [2]: 
```

Writing a maths function



```
def factorial(n):
    factorial=1
    for n in range(1,n+1):
       factorial=factorial*n
    return factorial

n=input('enter an integer value: ')
print n, ("factorial is"),factorial(n)
print type(factorial(n))

```

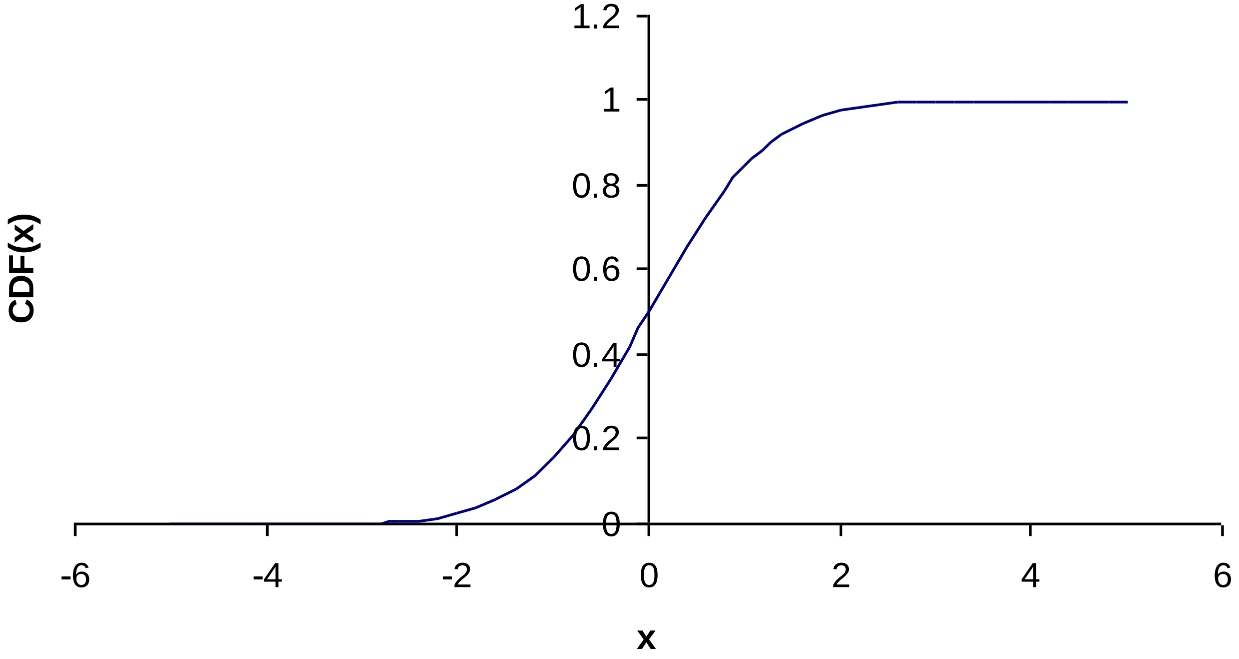
Python

```
In [42]: %run "c:\users\riaz\appdata\local\temp\tmppoyui8.py"

enter an integer value: 6
6 factorial is 720
<type 'int'>

In [43]: %run "c:\users\riaz\appdata\local\temp\tmpzjmg8t.py"

enter an integer value: 15
15 factorial is 1307674368000
<type 'long'>

In [44]:
```

Approximating a Cumulative Distribution Function (CDF)

A random variable $X \sim N(0,1)$ has CDF

$$N(x) = Pr(X < x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}s^2} ds$$



The Algorithm

We can approximate this improper integral by using the numerical scheme which is accurate to 6 decimal places

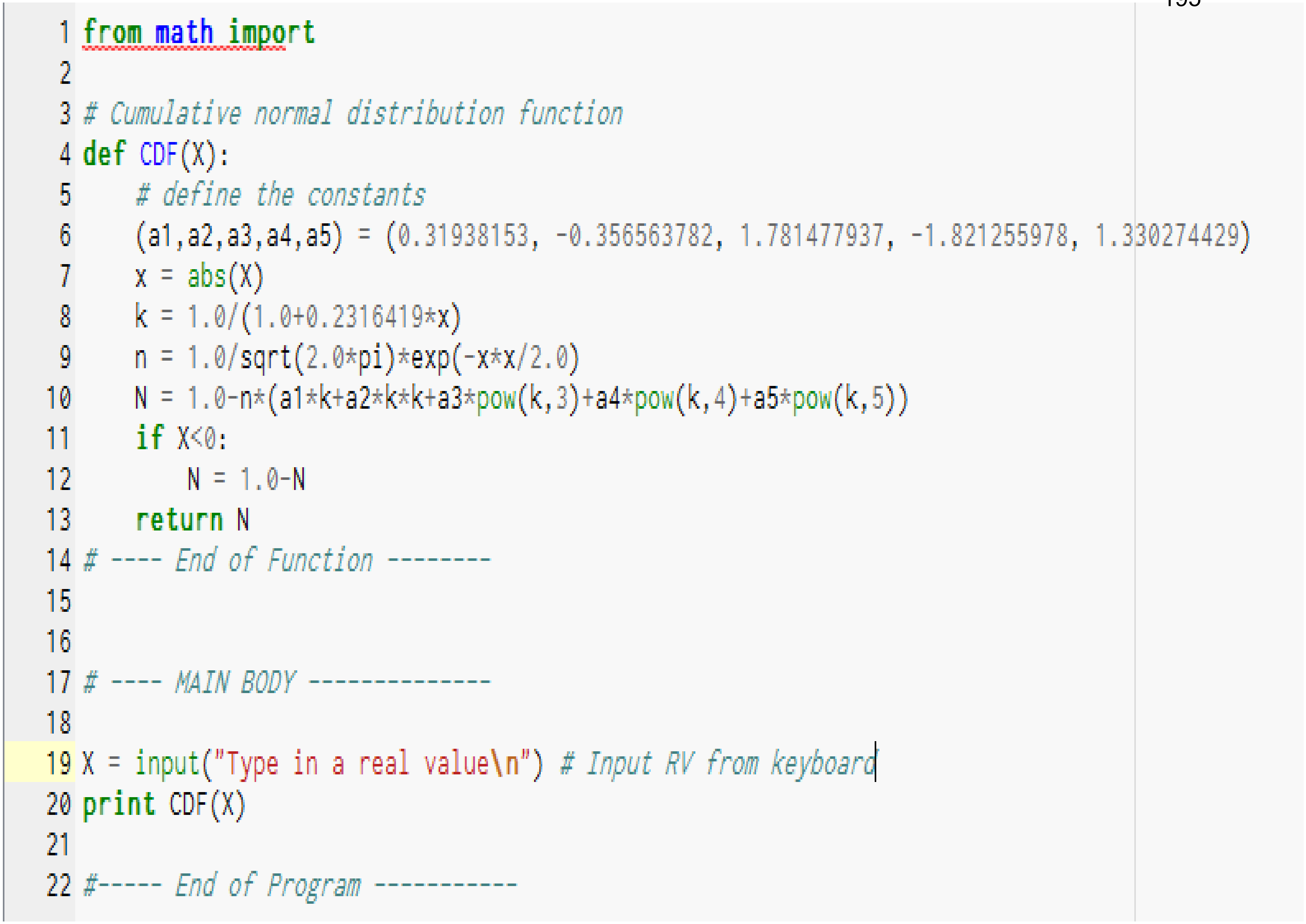
$$N(x) = \begin{cases} 1 - n(x)(a_1 k + a_2 k^2 + a_3 k^3 + a_4 k^4 + a_5 k^5) & x \geq 0 \\ 1 - N(-x) & x < 0 \end{cases}$$

Where

$$k = \frac{1}{1+0.2316419x} \quad \text{and}$$

a_1=0.319381530, a_2=−0.356563782, a_3=1.781477937
a_4=−1.821255978, a_5=1.330274429,

$$n(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

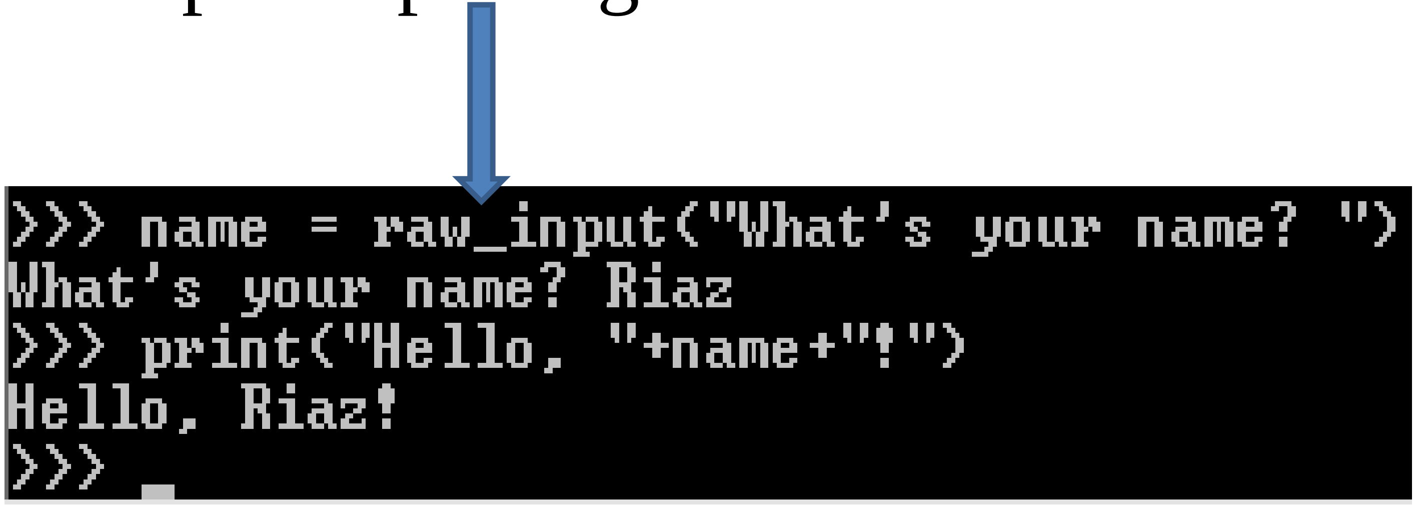


```
from math import

# Cumulative normal distribution function
def CDF(X):
    # define the constants
    (a1,a2,a3,a4,a5) = (0.31938153, -0.356563782, 1.781477937, -1.821255978, 1.330274429)
    x = abs(X)
    k = 1.0/(1.0+0.2316419*x)
    n = 1.0/sqrt(2.0*pi)*exp(-x*x/2.0)
    N = 1.0-n*(a1*k+a2*k*k+a3*pow(k,3)+a4*pow(k,4)+a5*pow(k,5))
    if X<0:
        N = 1.0-N
    return N
# ---- End of Function --------


# ---- MAIN BODY --------------

X = input("Type in a real value\n") # Input RV from keyboard
print CDF(X)

#----- End of Program -----------
```

Input from keyboard - revised

The function `raw_input()` can be used to request information from the user via the keyboard.

Example: inputting text

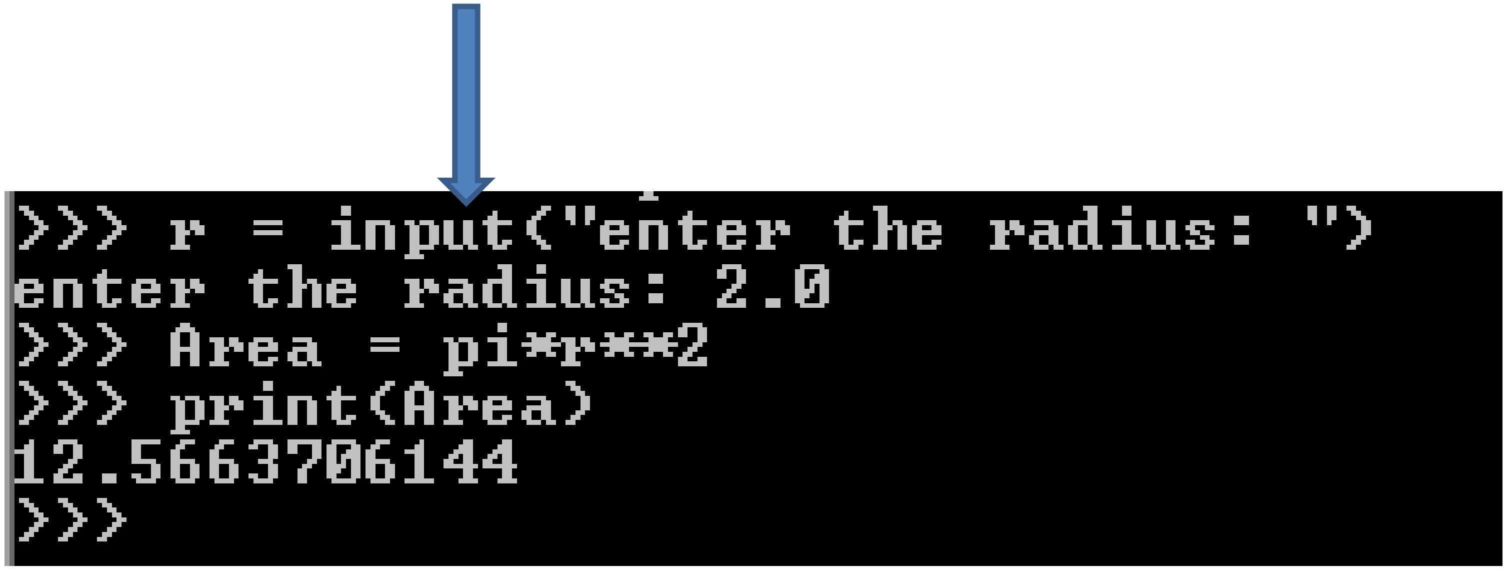


```
>>> name = raw_input("What's your name? ")
What's your name? Riaz
>>> print("Hello, "+name+"!")
Hello, Riaz!
>>> _
```

More keyboard input - numerical

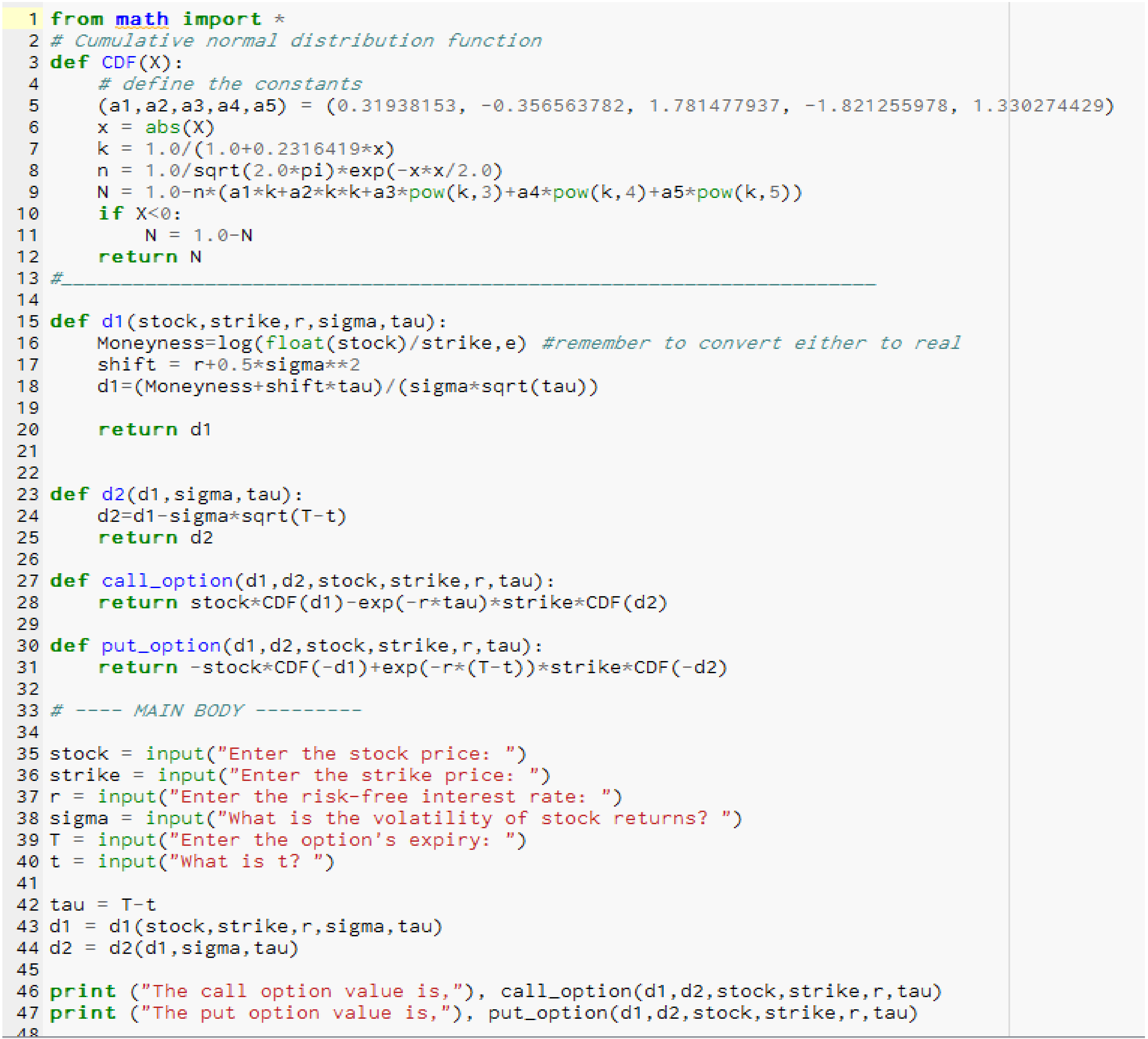
The function `input()` can be used to request information from the user via the keyboard.

Example: inputting numerical values



```
>>> r = input("enter the radius: ")
enter the radius: 2.0
>>> Area = pi*r**2
>>> print(Area)
12.5663706144
>>>
```

OPTION PRICER



```python
from math import *
# Cumulative normal distribution function
def CDF(X):
    # define the constants
    (a1,a2,a3,a4,a5) = (0.31938153, -0.356563782, 1.781477937, -1.821255978, 1.330274429)
    x = abs(X)
    k = 1.0/(1.0+0.2316419*x)
    n = 1.0/sqrt(2.0*pi)*exp(-x*x/2.0)
    N = 1.0-n*(a1*k+a2*k*k+a3*pow(k,3)+a4*pow(k,4)+a5*pow(k,5))
    if X<0:
        N = 1.0-N
    return N
#_____________________________________________________________________

def d1(stock,strike,r,sigma,tau):
    Moneyness=log(float(stock)/strike,e) #remember to convert either to real
    shift = r+0.5*sigma**2
    d1=(Moneyness+shift*tau)/(sigma*sqrt(tau))

    return d1


def d2(d1,sigma,tau):
    d2=d1-sigma*sqrt(T-t)
    return d2

def call_option(d1,d2,stock,strike,r,tau):
    return stock*CDF(d1)-exp(-r*tau)*strike*CDF(d2)

def put_option(d1,d2,stock,strike,r,tau):
    return -stock*CDF(-d1)+exp(-r*(T-t))*strike*CDF(-d2)

# ---- MAIN BODY ---------

stock = input("Enter the stock price: ")
strike = input("Enter the strike price: ")
r = input("Enter the risk-free interest rate: ")
sigma = input("What is the volatility of stock returns? ")
T = input("Enter the option's expiry: ")
t = input("What is t? ")

tau = T-t
d1 = d1(stock,strike,r,sigma,tau)
d2 = d2(d1,sigma,tau)

print ("The call option value is,"), call_option(d1,d2,stock,strike,r,tau)
print ("The put option value is,"), put_option(d1,d2,stock,strike,r,tau)
```

Numpy

The numpy package is used in almost all numerical computation using Python. The package provides high-performance vector, matrix and higher-dimensional data structures for Python. Its flagship object is the powerful N – dimensional array

```
>>> from numpy import*
```

You can also use one or more of:

```
>>> from numpy.linalg import*
```

```
>>> from numpy.fft import*
```

```
>>> from numpy.random import*
```

and others

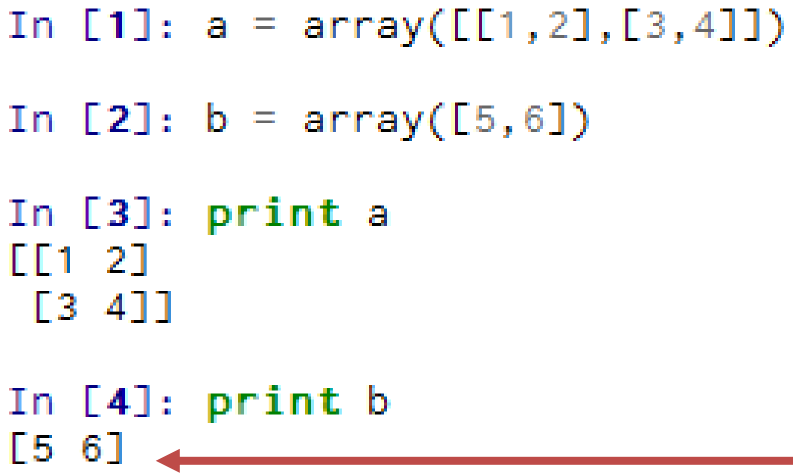
Arrays

The most basic numpy data type.

Matrices are specialised 2-D arrays.

Types int, float, complex forms available

Example: $\boldsymbol{a} = \begin{bmatrix} \mathbf{1} & \mathbf{2} \\ \mathbf{3} & \mathbf{4} \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} \mathbf{5} \\ \mathbf{6} \end{bmatrix}$



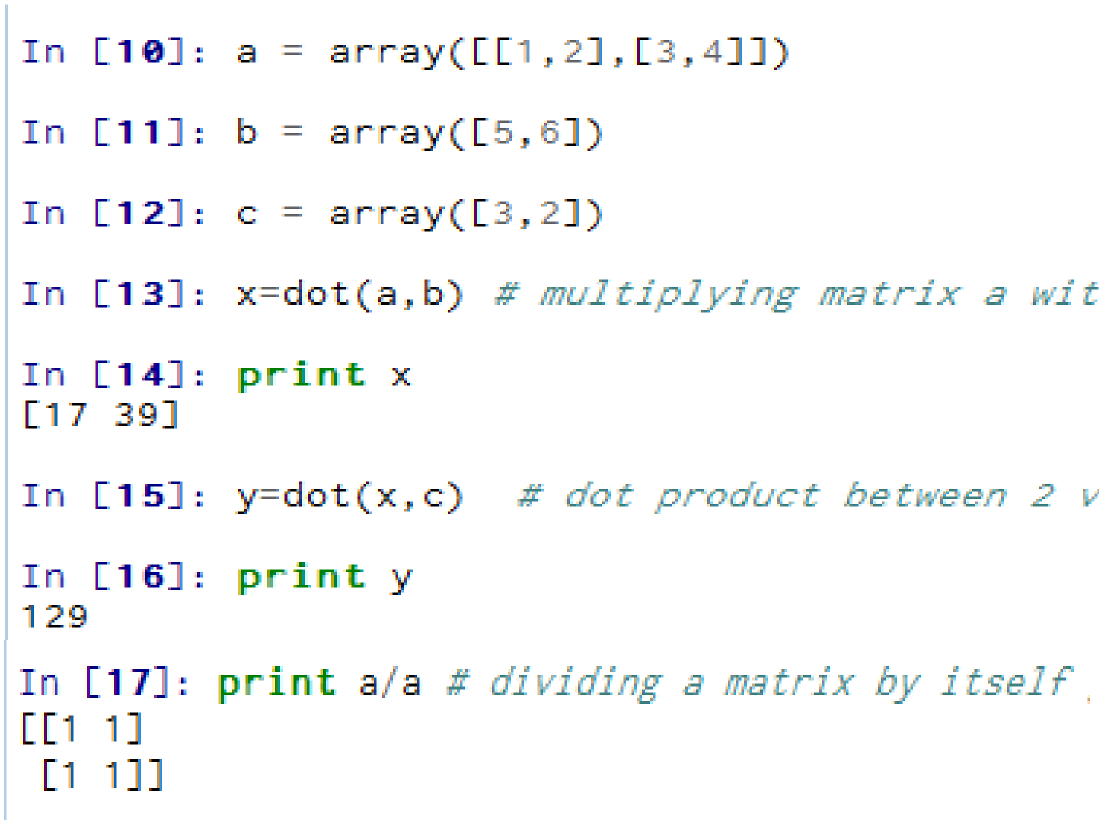
```
In [1]: a = array([[1,2],[3,4]])

In [2]: b = array([5,6])

In [3]: print a
[[1 2]
 [3 4]]

In [4]: print b
[5 6]
```

Vector displayed
in row form

Simple operations on arrays

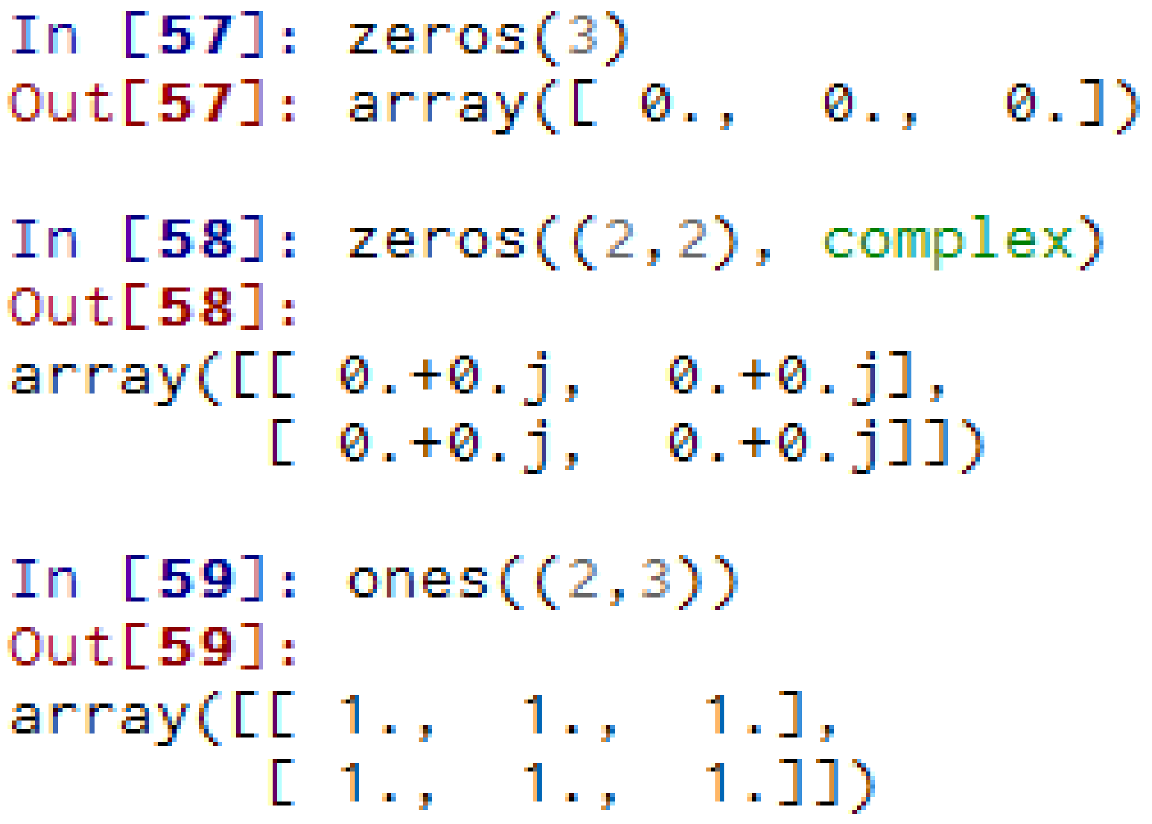
$$\boldsymbol{a} = \begin{bmatrix} \mathbf{1} & \mathbf{2} \\ \mathbf{3} & \mathbf{4} \end{bmatrix},\ \ \boldsymbol{b} = \begin{bmatrix} \mathbf{5} \\ \mathbf{6} \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} \mathbf{3} \\ \mathbf{2} \end{bmatrix}$$

Example: $\boldsymbol{d} = \boldsymbol{a} \times \boldsymbol{b} = \begin{bmatrix} \mathbf{17} \\ \mathbf{39} \end{bmatrix}$; $\boldsymbol{d} \cdot \boldsymbol{c} = [\mathbf{17}, \mathbf{39}] \cdot [\mathbf{3}, \mathbf{2}]$=129



```
In [10]: a = array([[1,2],[3,4]])

In [11]: b = array([5,6])

In [12]: c = array([3,2])

In [13]: x=dot(a,b) # multiplying matrix a with vector b

In [14]: print x
[17 39]

In [15]: y=dot(x,c)  # dot product between 2 vectors

In [16]: print y
129

In [17]: print a/a # dividing a matrix by itself
[[1 1]
 [1 1]]
```

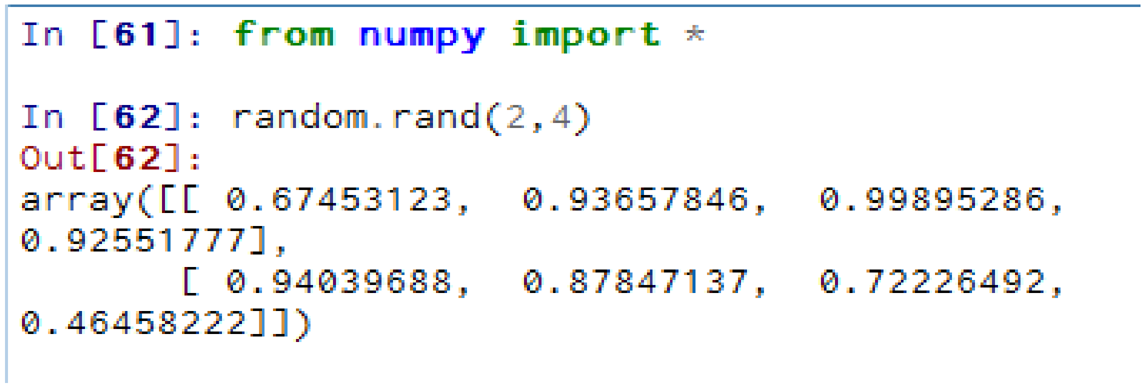
Filling arrays with identical elements



```
In [57]: zeros(3)
Out[57]: array([ 0.,  0.,  0.])

In [58]: zeros((2,2), complex)
Out[58]:
array([[ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j]])

In [59]: ones((2,3))
Out[59]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

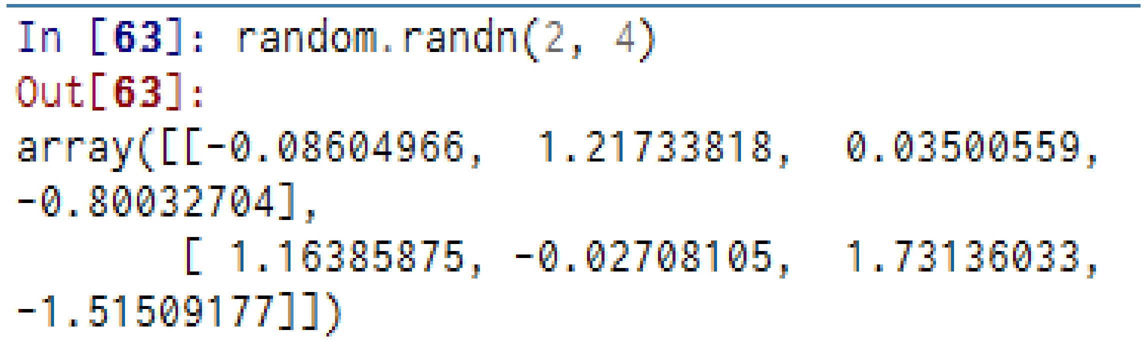
Filling arrays with random numbers

`rand`: random numbers uniformly distributed between 0 and 1



```
In [61]: from numpy import *

In [62]: random.rand(2,4)
Out[62]:
array([[ 0.67453123,  0.93657846,  0.99895286,
0.92551777],
       [ 0.94039688,  0.87847137,  0.72226492,
0.46458222]])
```

`nrand`: Normal (Gaussian) distribution $N(0,1)$



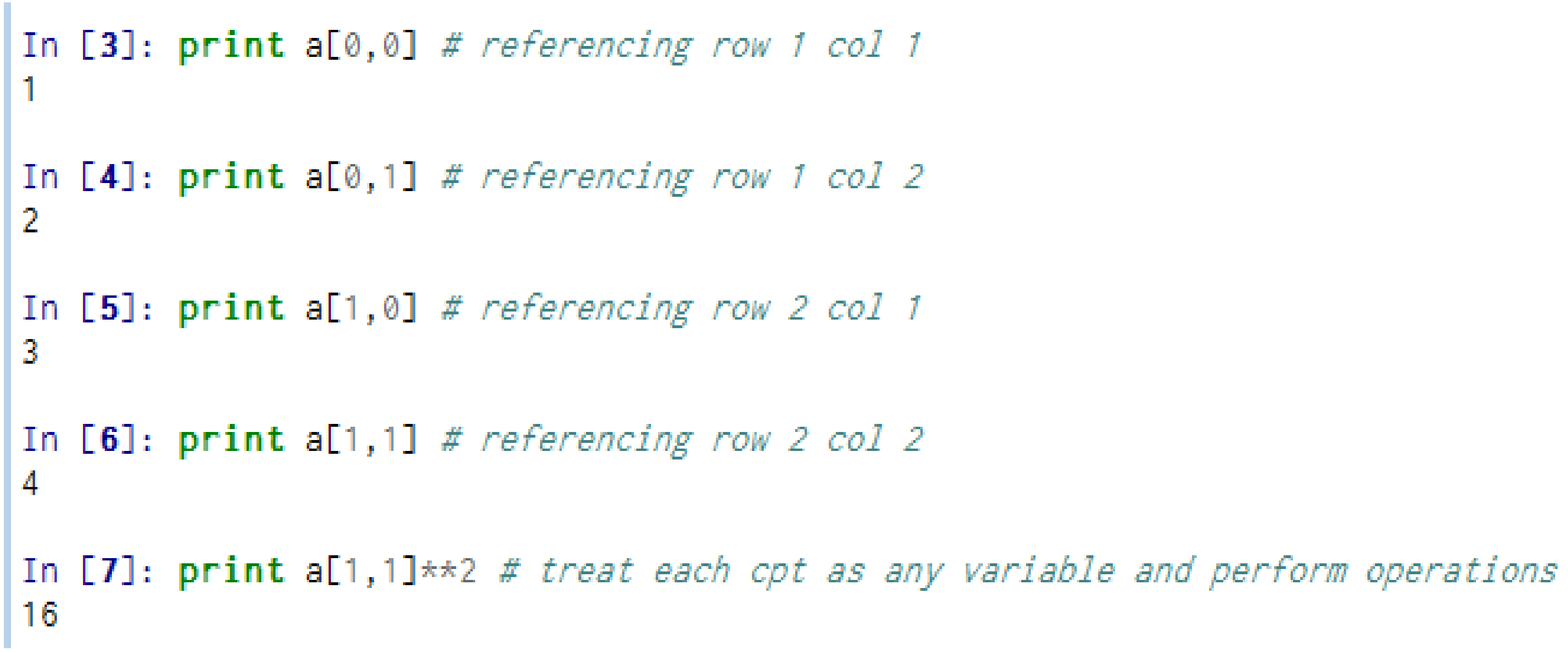
```
In [63]: random.randn(2, 4)
Out[63]:
array([[-0.08604966,  1.21733818,  0.03500559,
-0.80032704],
       [ 1.16385875, -0.02708105,  1.73136033,
-1.51509177]])
```

Other standard distributions are also available

Indexing starts at zero!

$$\boldsymbol{b} = \begin{bmatrix} \boldsymbol{b_0} \\ \\ \boldsymbol{b_1} \end{bmatrix},$$ A vector component $\boldsymbol{b_i}$ can be accessed in python as `b[i]`

$$\boldsymbol{a} = \begin{bmatrix} \boldsymbol{a_{00}} & \boldsymbol{a_{01}} \\ \\ \boldsymbol{a_{10}} & \boldsymbol{a_{11}} \end{bmatrix},$$ A component $\boldsymbol{a_{ij}}$ can be accessed in python as `a[i,j]`



```
In [3]: print a[0,0] # referencing row 1 col 1
1

In [4]: print a[0,1] # referencing row 1 col 2
2

In [5]: print a[1,0] # referencing row 2 col 1
3

In [6]: print a[1,1] # referencing row 2 col 2
4

In [7]: print a[1,1]**2 # treat each cpt as any variable and perform operations
16
```

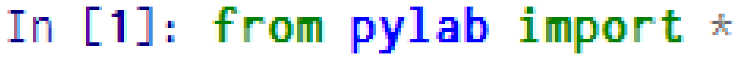
matplotlib – 2D and 3D plotting

matplotlib.pyplot is an amazing 2D and 3D graphics library containing a collection of command line style functions that make matplotlib work like MATLAB. The advantages of using this library include:

- As with Python, easy to get started
- Support for $\LaTeX$ formatted labels and texts
- Superior levels of control of all aspects of high quality figures in many formats (e.g. PNG, PDF, SVG, EPS, PGF)

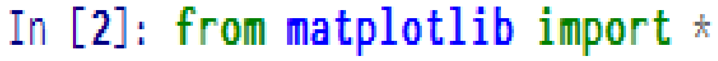
matplotlib – getting started

To get started, the easy way to include matplotlib in a Python program is with:



```
In [1]: from pylab import *
```

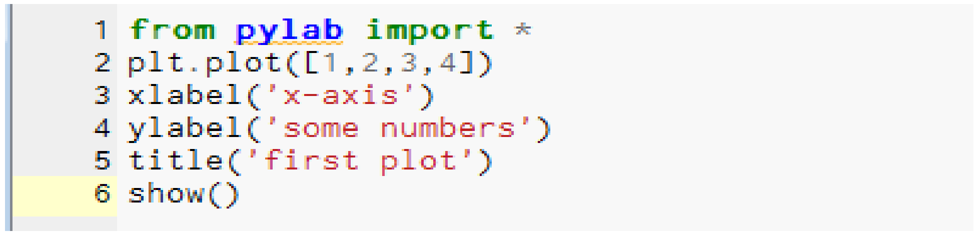
Can also have



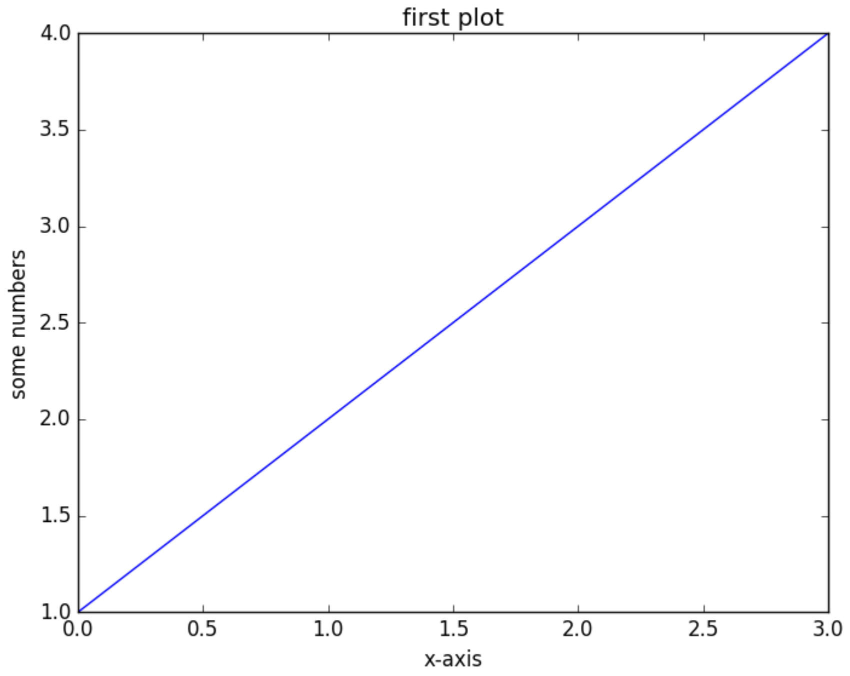
```
In [2]: from matplotlib import *
```

The advantage of using matplotlib is being able to draw MATLAB still graphs.

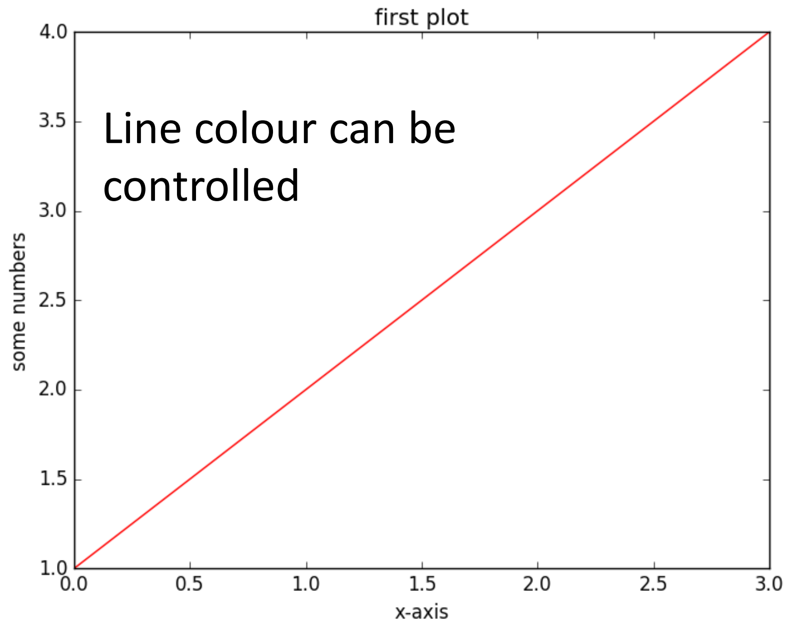
First Plot



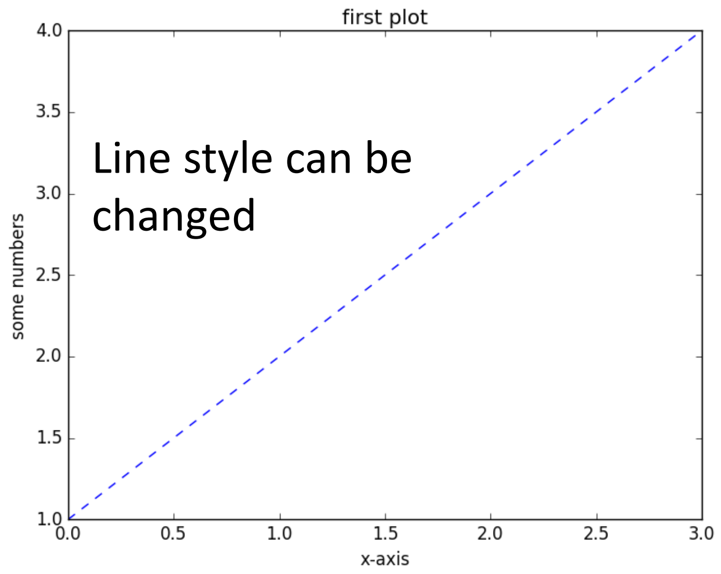
```
from pylab import *
plt.plot([1,2,3,4])
xlabel('x-axis')
ylabel('some numbers')
title('first plot')
show()
```



plot([1,2,3,4],'r')



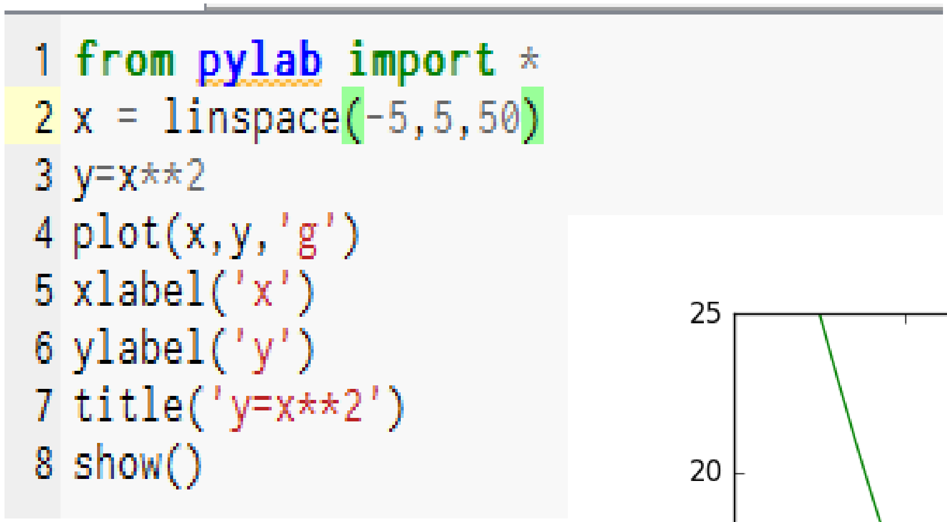
Line colour can be controlled



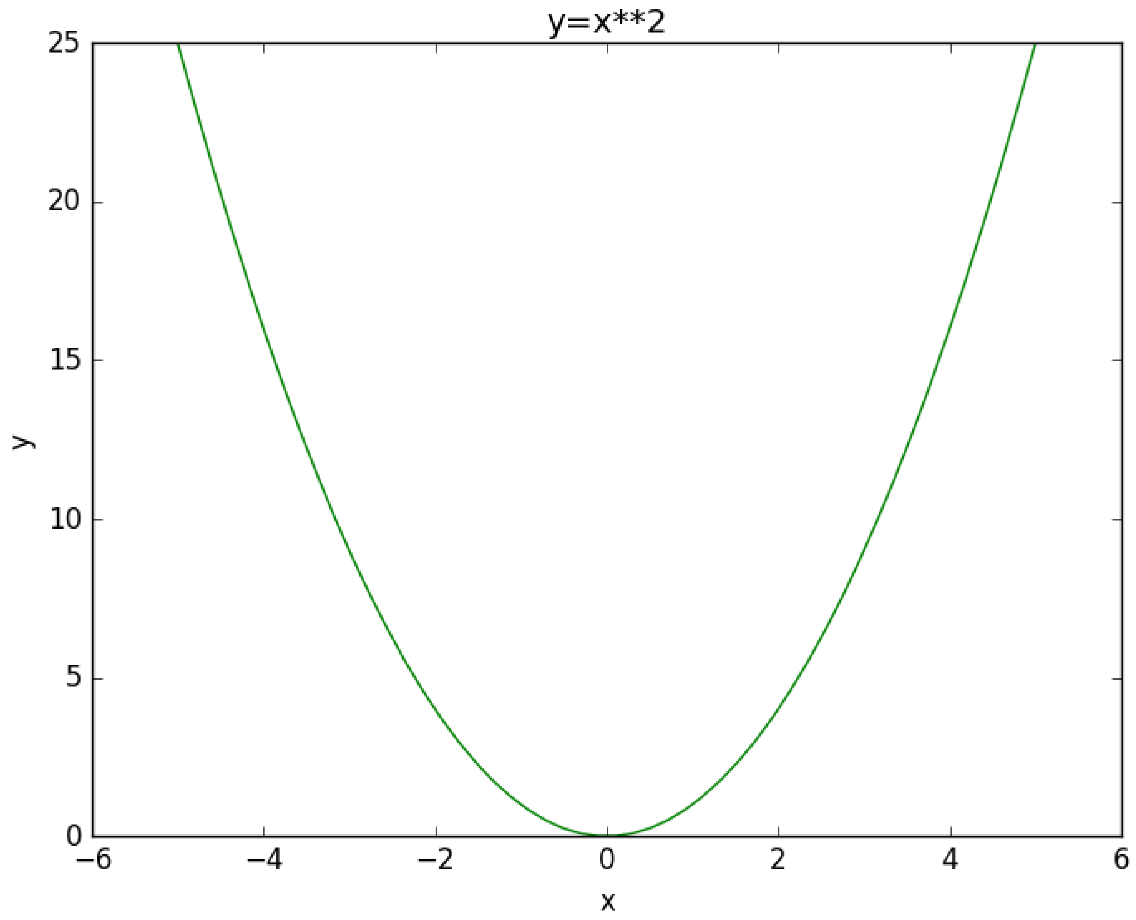
Line style can be changed

plot([1,2,3,4],'b--')

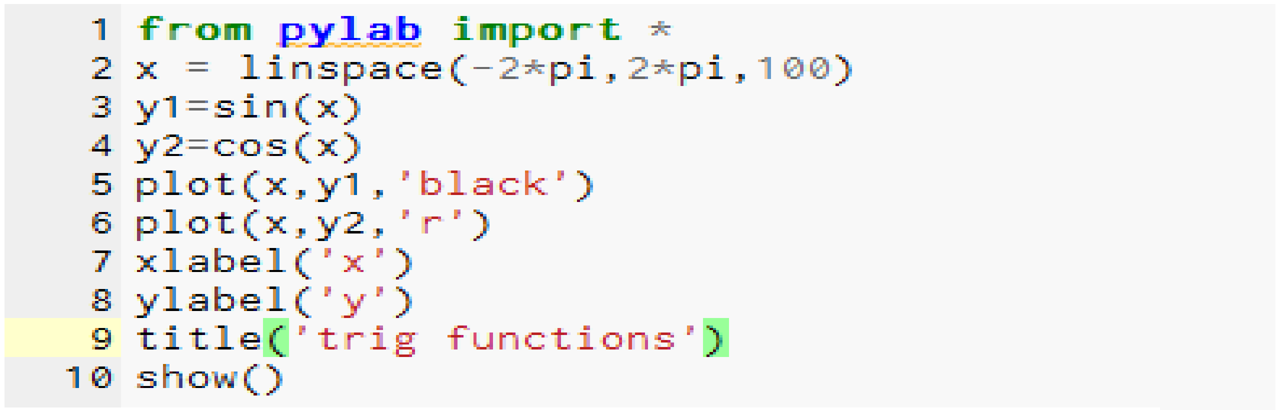
Nonlinear Plot - Quadratic



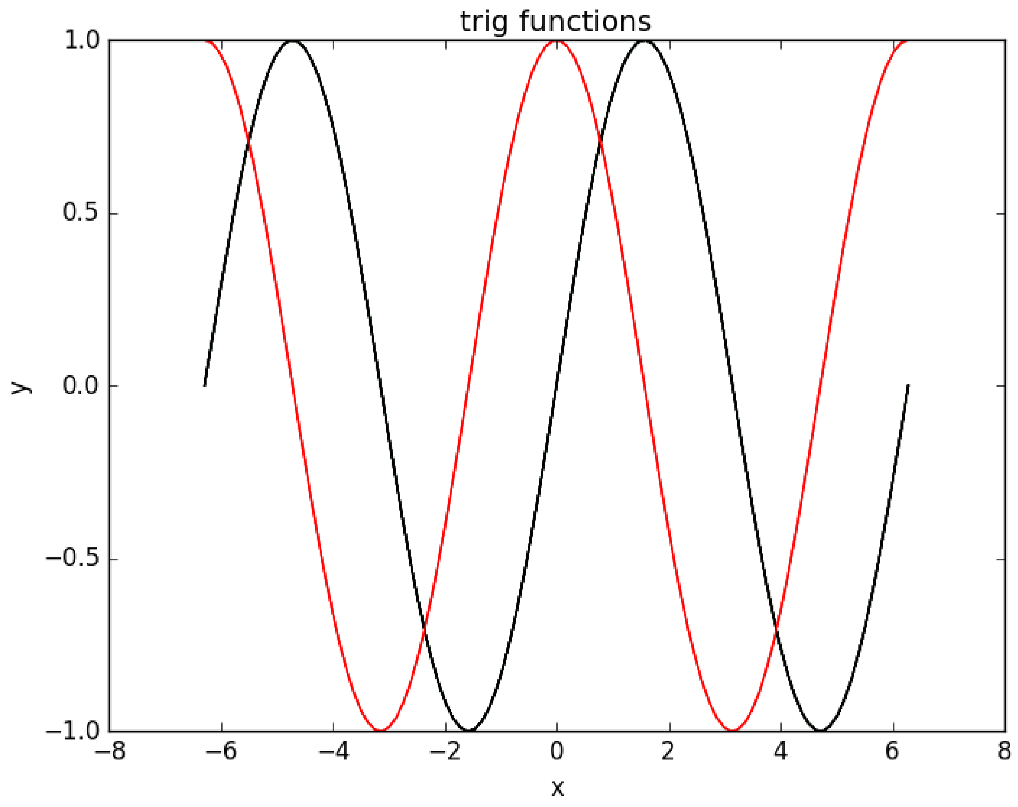
```python
from pylab import *
x = linspace(-5,5,50)
y=x**2
plot(x,y,'g')
xlabel('x')
ylabel('y')
title('y=x**2')
show()
```



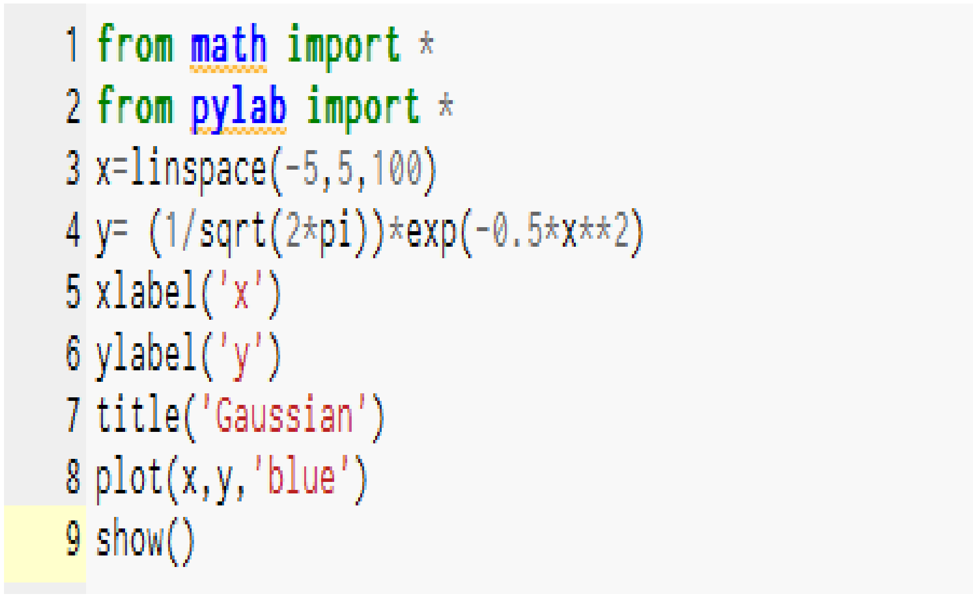
Nonlinear Plot – Trig functions



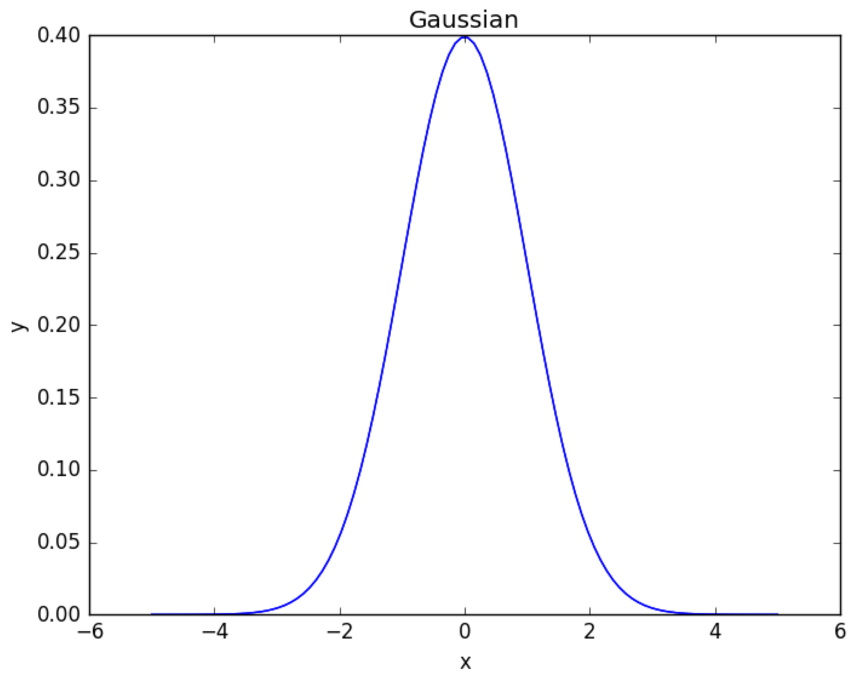
```python
from pylab import *
x = linspace(-2*pi,2*pi,100)
y1=sin(x)
y2=cos(x)
plot(x,y1,'black')
plot(x,y2,'r')
xlabel('x')
ylabel('y')
title('trig functions')
show()
```



Plotting a Gaussian

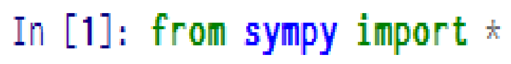


```
from math import *
from pylab import *
x=linspace(-5,5,100)
y= (1/sqrt(2*pi))*exp(-0.5*x**2)
xlabel('x')
ylabel('y')
title('Gaussian')
plot(x,y,'blue')
show()
```



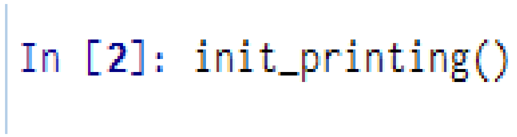
Sympy – Symbolic algebra in Python

Sympy is one of two Computer Algebra Systems (CAS) for Python. To get started, import the module `sympy`:



```
In [1]: from sympy import *
```

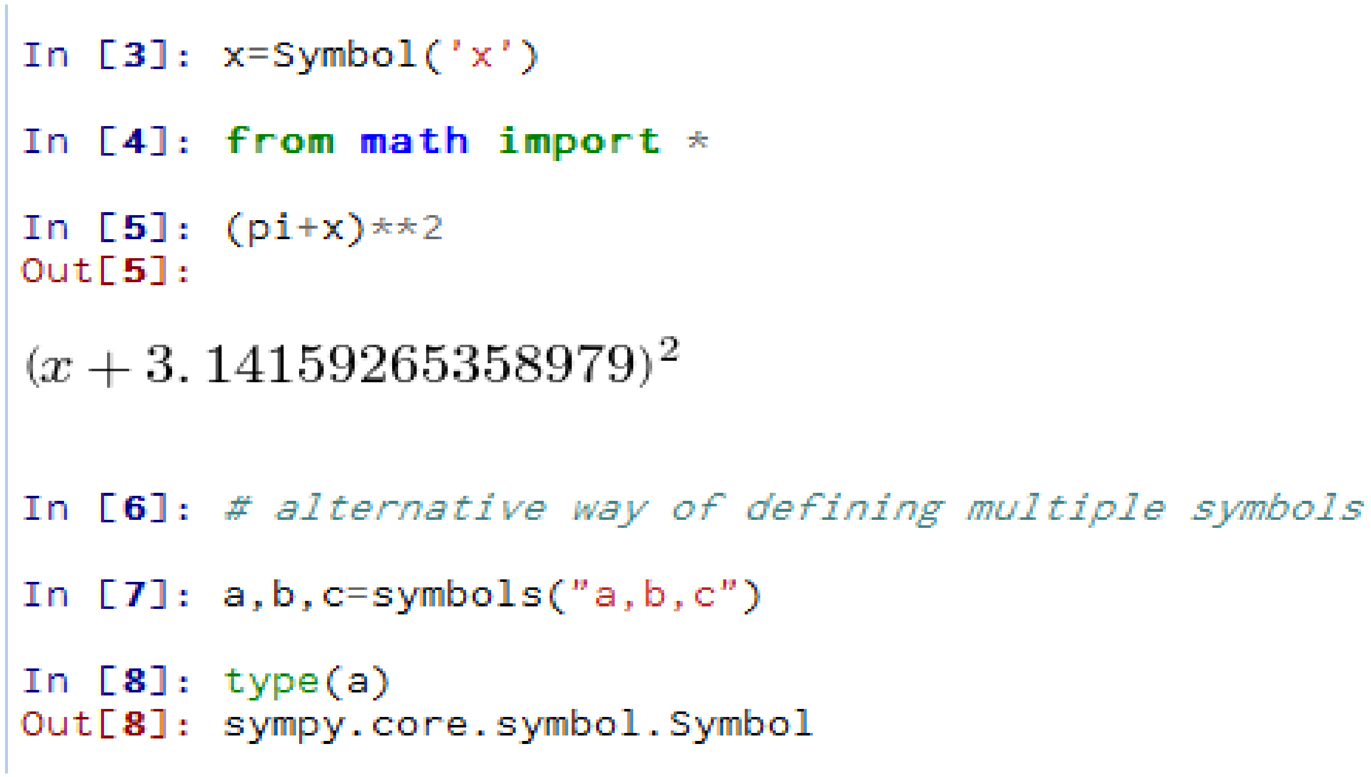
To get beautiful LaTeX formatted output, simply run:



```
In [2]: init_printing()
```

Symbolic Variables: In SymPy we need to create symbols for the variables we want to work with. This can be done using the **`Symbol`** class.

Use of Symbolic Variables

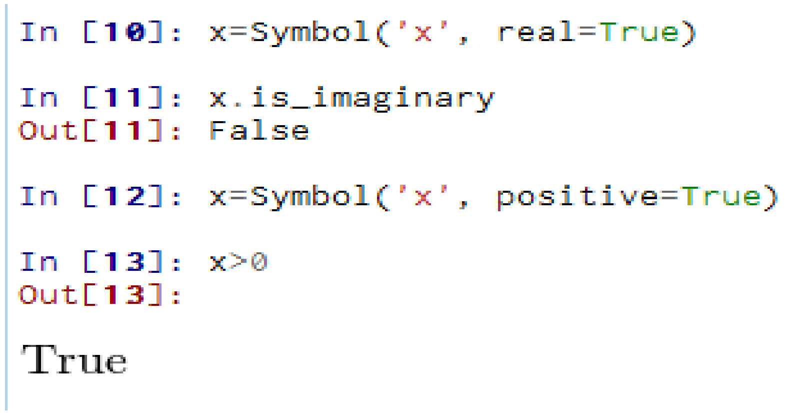


```
In [3]: x=Symbol('x')

In [4]: from math import *

In [5]: (pi+x)**2
Out[5]:
```

$$(x + 3.14159265358979)^2$$

```
In [6]: # alternative way of defining multiple symbols

In [7]: a,b,c=symbols("a,b,c")

In [8]: type(a)
Out[8]: sympy.core.symbol.Symbol
```

We can add assumptions to variables when we create them:

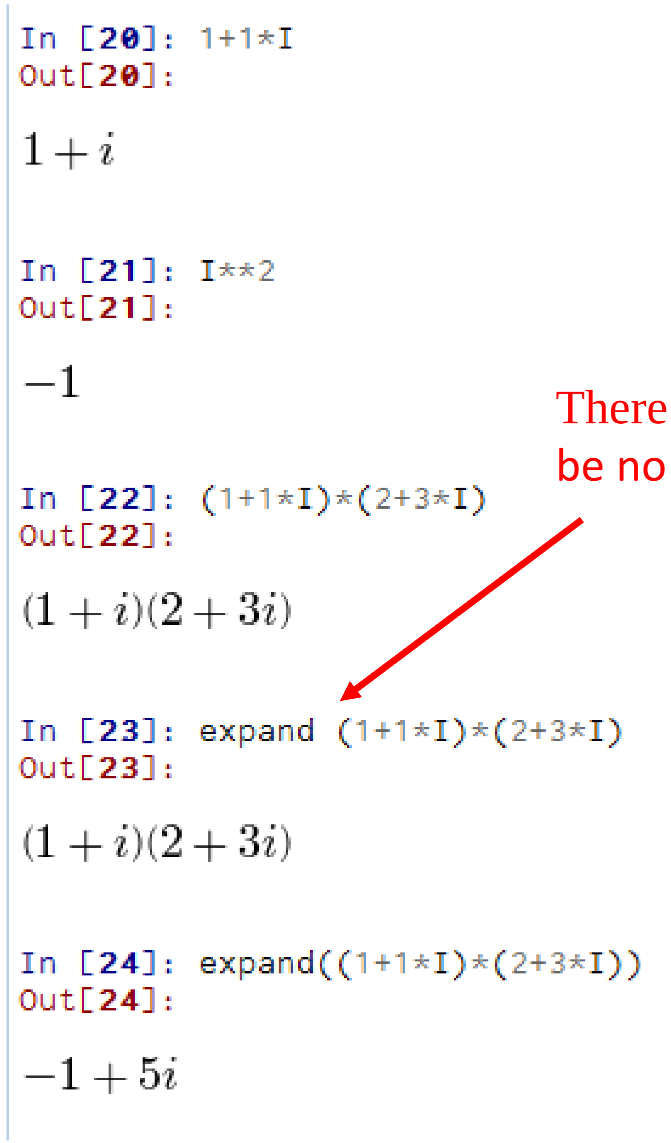


```
In [10]: x=Symbol('x', real=True)

In [11]: x.is_imaginary
Out[11]: False

In [12]: x=Symbol('x', positive=True)

In [13]: x>0
Out[13]:
```

$$\mathrm{True}$$

Complex Numbers

The imaginary unit $i = \sqrt{-1}$ denoted `I` in SymPy:



```
In [20]: 1+1*I
Out[20]:
```

$$1 + i$$

```
In [21]: I**2
Out[21]:
```

$$-1$$

```
In [22]: (1+1*I)*(2+3*I)
Out[22]:
```

$$(1 + i)(2 + 3i)$$

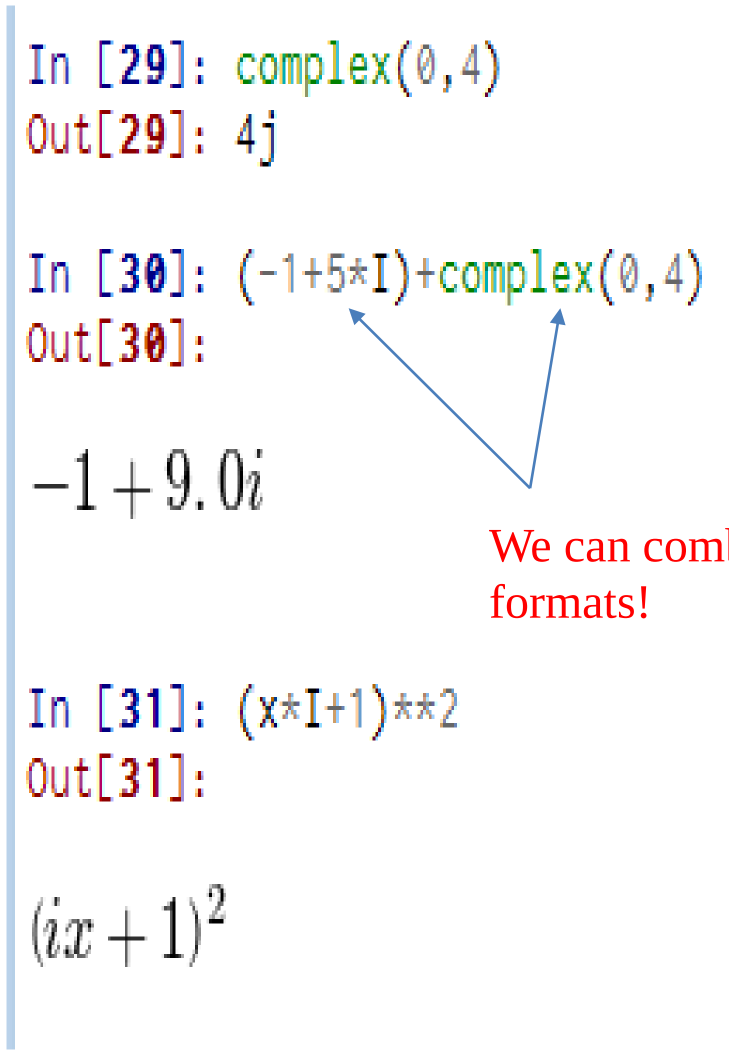
```
In [23]: expand (1+1*I)*(2+3*I)
Out[23]:
```

$$(1 + i)(2 + 3i)$$

```
In [24]: expand((1+1*I)*(2+3*I))
Out[24]:
```

$$-1 + 5i$$

There should be no space



```
In [29]: complex(0,4)
Out[29]: 4j

In [30]: (-1+5*I)+complex(0,4)
Out[30]:
```

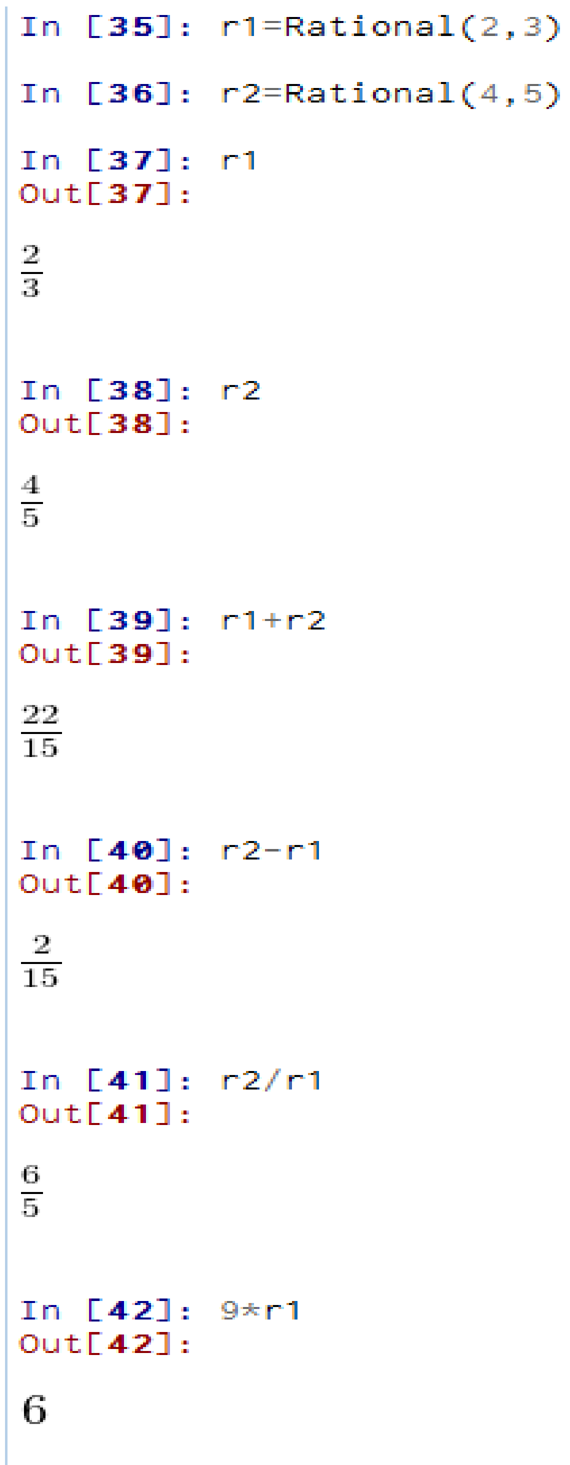
$$-1 + 9.0i$$

We can combine formats!

```
In [31]: (x*I+1)**2
Out[31]:
```

$$(ix + 1)^2$$

Rational Numbers



```
In [35]: r1=Rational(2,3)

In [36]: r2=Rational(4,5)

In [37]: r1
Out[37]:
```

$$\frac{2}{3}$$

```
In [38]: r2
Out[38]:
```

$$\frac{4}{5}$$

```
In [39]: r1+r2
Out[39]:
```

$$\frac{22}{15}$$

```
In [40]: r2-r1
Out[40]:
```

$$\frac{2}{15}$$

```
In [41]: r2/r1
Out[41]:
```

$$\frac{6}{5}$$

```
In [42]: 9*r1
Out[42]:
```

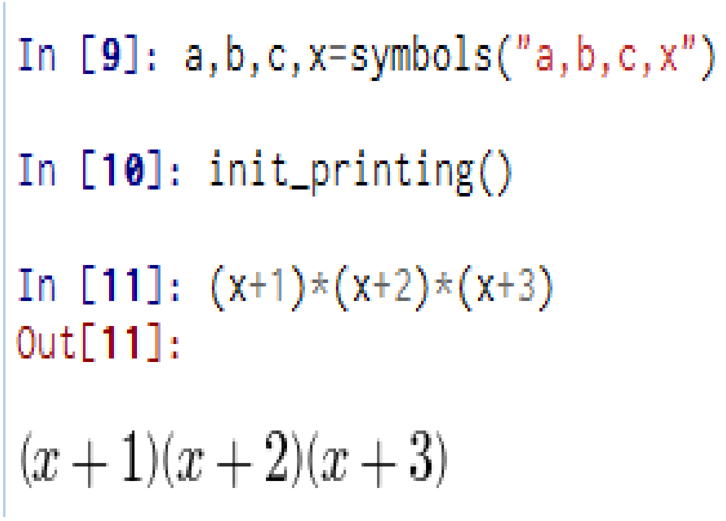
$$6$$

There are three different numerical types in SymPy: **`Real`**, **`Rational`**, **`Integer`**:

Algebraic Manipulations

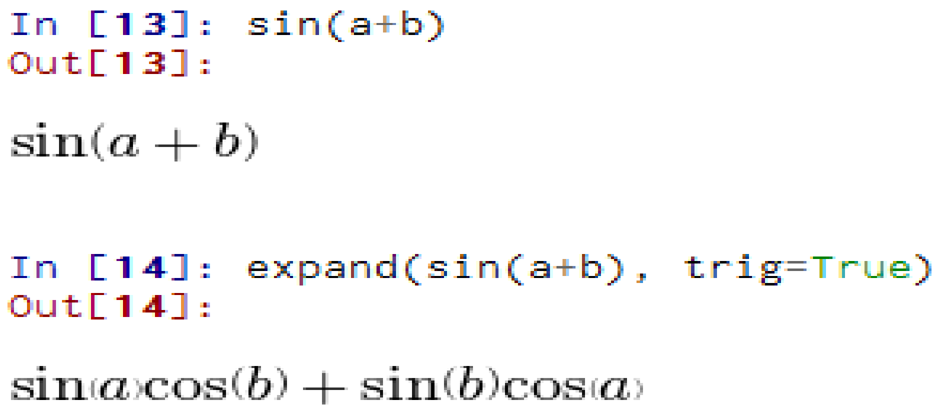
A main use of CAS is to perform algebraic manipulations of expressions. For example, we may wish to expand a product, factor an expression, or simplify an expression. The functions for doing these basic operations in SymPy are demonstrated below:

- **Expand and factor**



```
In [9]: a,b,c,x=symbols("a,b,c,x")

In [10]: init_printing()

In [11]: (x+1)*(x+2)*(x+3)
Out[11]:
```

$$(x+1)(x+2)(x+3)$$

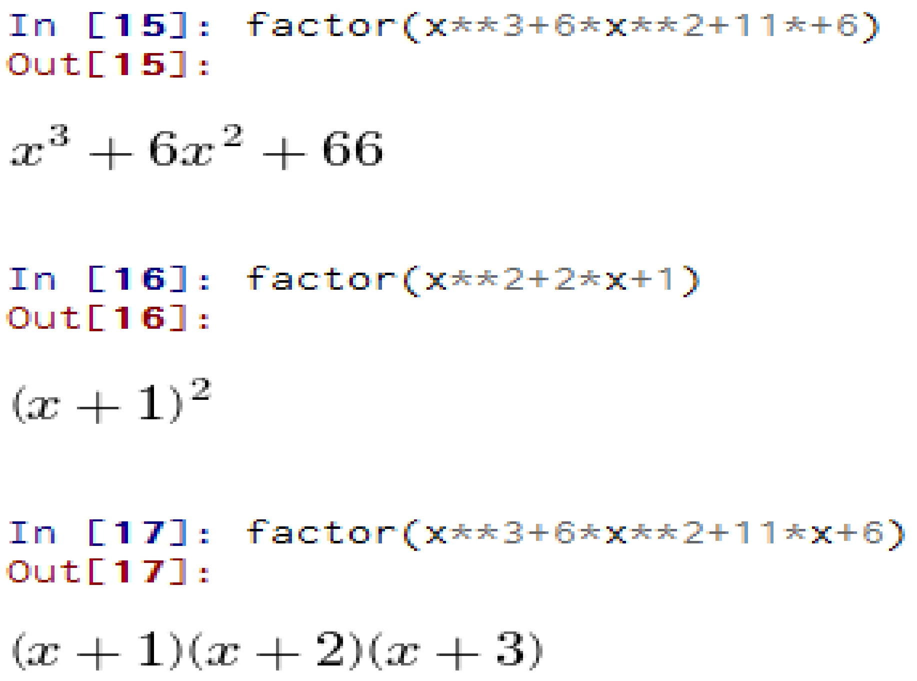


```
In [13]: sin(a+b)
Out[13]:
```

$$\sin(a+b)$$

```
In [14]: expand(sin(a+b), trig=True)
Out[14]:
```

$$\sin(a)\cos(b)+\sin(b)\cos(a)$$



```
In [15]: factor(x**3+6*x**2+11*+6)
Out[15]:
```

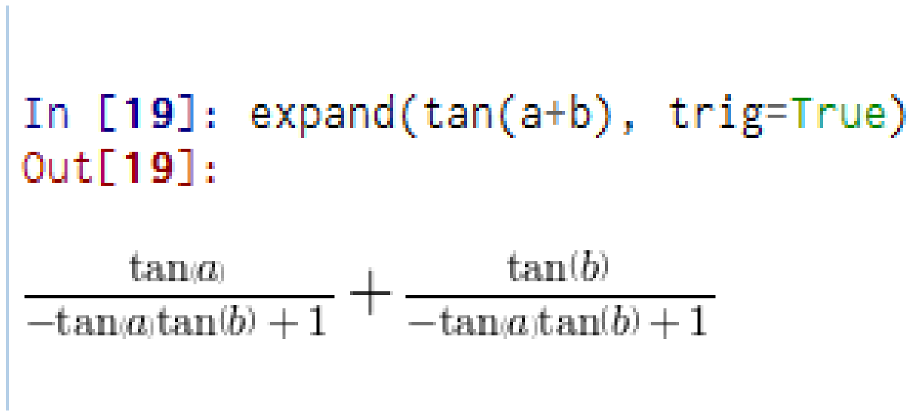
$$x^3+6x^2+66$$

```
In [16]: factor(x**2+2*x+1)
Out[16]:
```

$$(x+1)^2$$

```
In [17]: factor(x**3+6*x**2+11*x+6)
Out[17]:
```

$$(x+1)(x+2)(x+3)$$

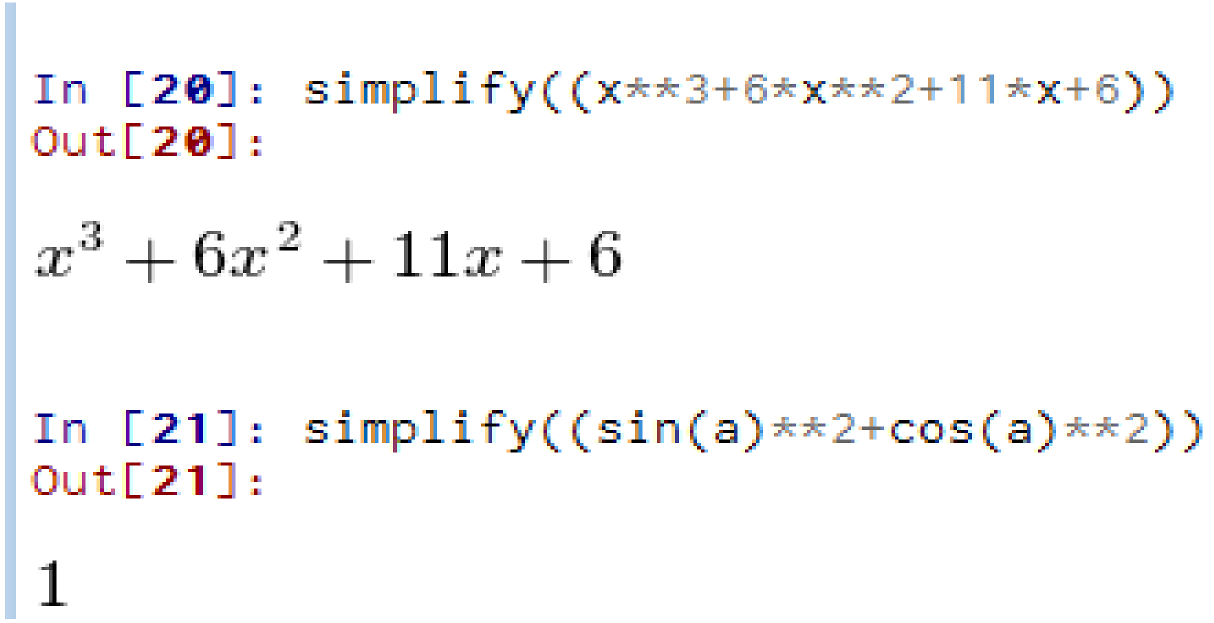


```
In [19]: expand(tan(a+b), trig=True)
Out[19]:
```

$$\frac{\tan(a)}{-\tan(a)\tan(b)+1}+\frac{\tan(b)}{-\tan(a)\tan(b)+1}$$

Simplify

The simplify function attempts to simplify an expression into a set of nicer looking smaller terms using various techniques. More specific simplification alternatives to the simplify functions also exist: `trigsimp`, `powsimp`, `logcombine`, etc

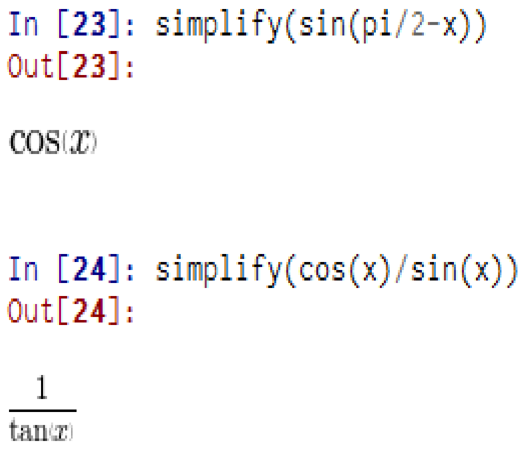


```
In [20]: simplify((x**3+6*x**2+11*x+6))
Out[20]:
```

$$x^3 + 6x^2 + 11x + 6$$

```
In [21]: simplify((sin(a)**2+cos(a)**2))
Out[21]:
```

$$1$$



```
In [23]: simplify(sin(pi/2-x))
Out[23]:
```

$$\cos(x)$$

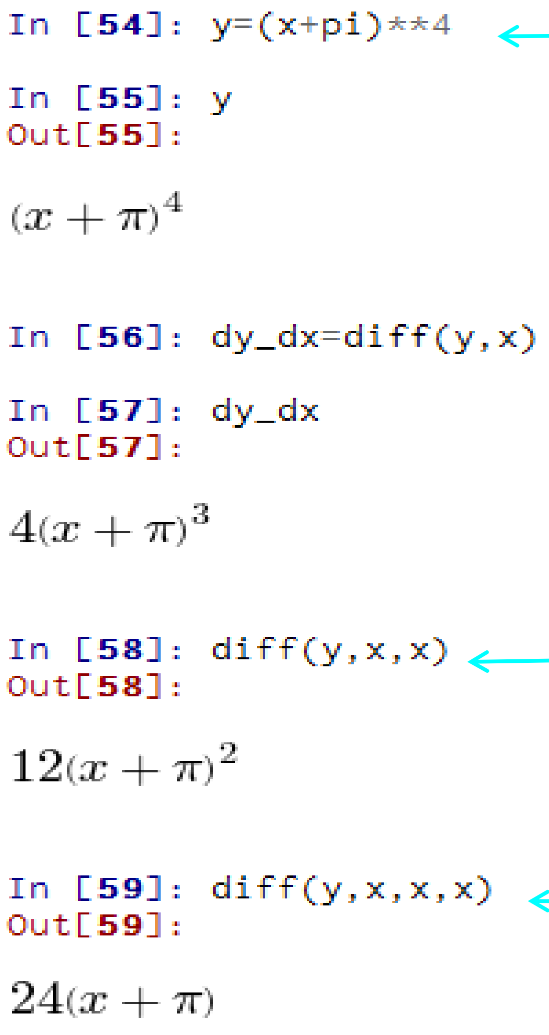
```
In [24]: simplify(cos(x)/sin(x))
Out[24]:
```

$$\frac{1}{\tan(x)}$$

Calculus I – Differentiation I

A powerful feature of CAS is its Calculus functionality like derivatives and integrals of algebraic expressions

➢ **Differentiation –** Use the diff function. The first argument is the expression to take the derivative of, and the second is the symbol by which to take the derivative:



```
In [54]: y=(x+pi)**4

In [55]: y
Out[55]:
```

$$(x+\pi)^4$$

```
In [56]: dy_dx=diff(y,x)

In [57]: dy_dx
Out[57]:
```

$$4(x+\pi)^3$$

```
In [58]: diff(y,x,x)
Out[58]:
```

$$12(x+\pi)^2$$

```
In [59]: diff(y,x,x,x)
Out[59]:
```

$$24(x+\pi)$$

New function defined

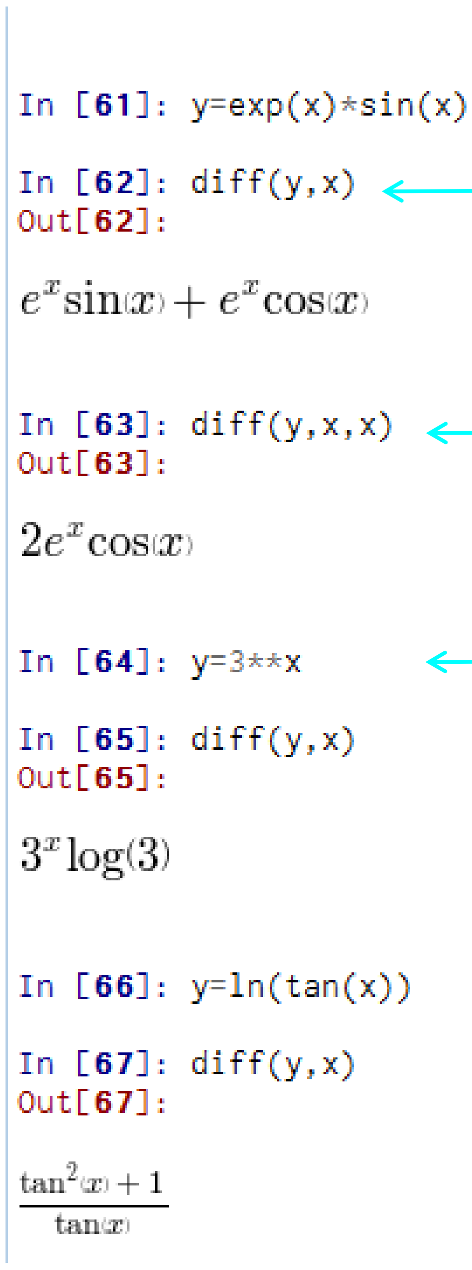
Differentiate *y* wrt *x* once

Differentiate *y* wrt *x* twice

Differentiate *y* wrt *x* three times

Calculus I – Differentiation II

Trig functions and transcendental functions can also be differentiated



```
In [61]: y=exp(x)*sin(x)

In [62]: diff(y,x)
Out[62]:
```

$e^x \sin(x) + e^x \cos(x)$

```
In [63]: diff(y,x,x)
Out[63]:
```

$2e^x \cos(x)$

```
In [64]: y=3**x

In [65]: diff(y,x)
Out[65]:
```

$3^x \log(3)$

```
In [66]: y=ln(tan(x))

In [67]: diff(y,x)
Out[67]:
```

$\frac{\tan^2(x) + 1}{\tan(x)}$

New function defined

Differentiate y wrt x once

Differentiate y twice wrt x

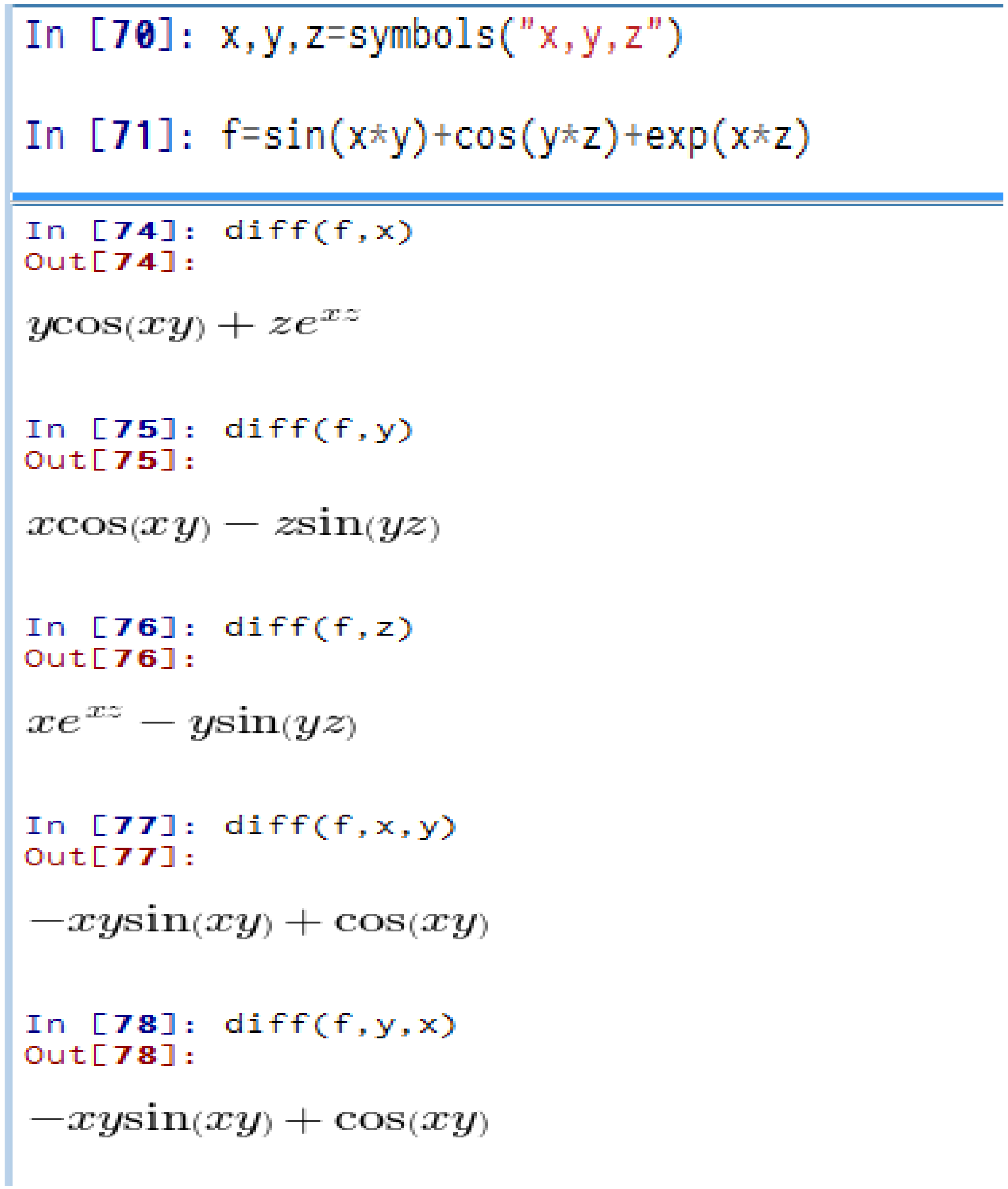
New function defined

$$\frac{d^n y}{dx^n} = \text{diff}\,(\mathtt{y,x,n})$$

where `n` is the order of differentiation

Calculus I – Differentiation III

We can do partial differentiation on multivariate functions



```
In [70]: x,y,z=symbols("x,y,z")

In [71]: f=sin(x*y)+cos(y*z)+exp(x*z)

In [74]: diff(f,x)
Out[74]:
```

$$y\cos(xy) + ze^{xz}$$

```
In [75]: diff(f,y)
Out[75]:
```

$$x\cos(xy) - z\sin(yz)$$

```
In [76]: diff(f,z)
Out[76]:
```

$$xe^{xz} - y\sin(yz)$$

```
In [77]: diff(f,x,y)
Out[77]:
```

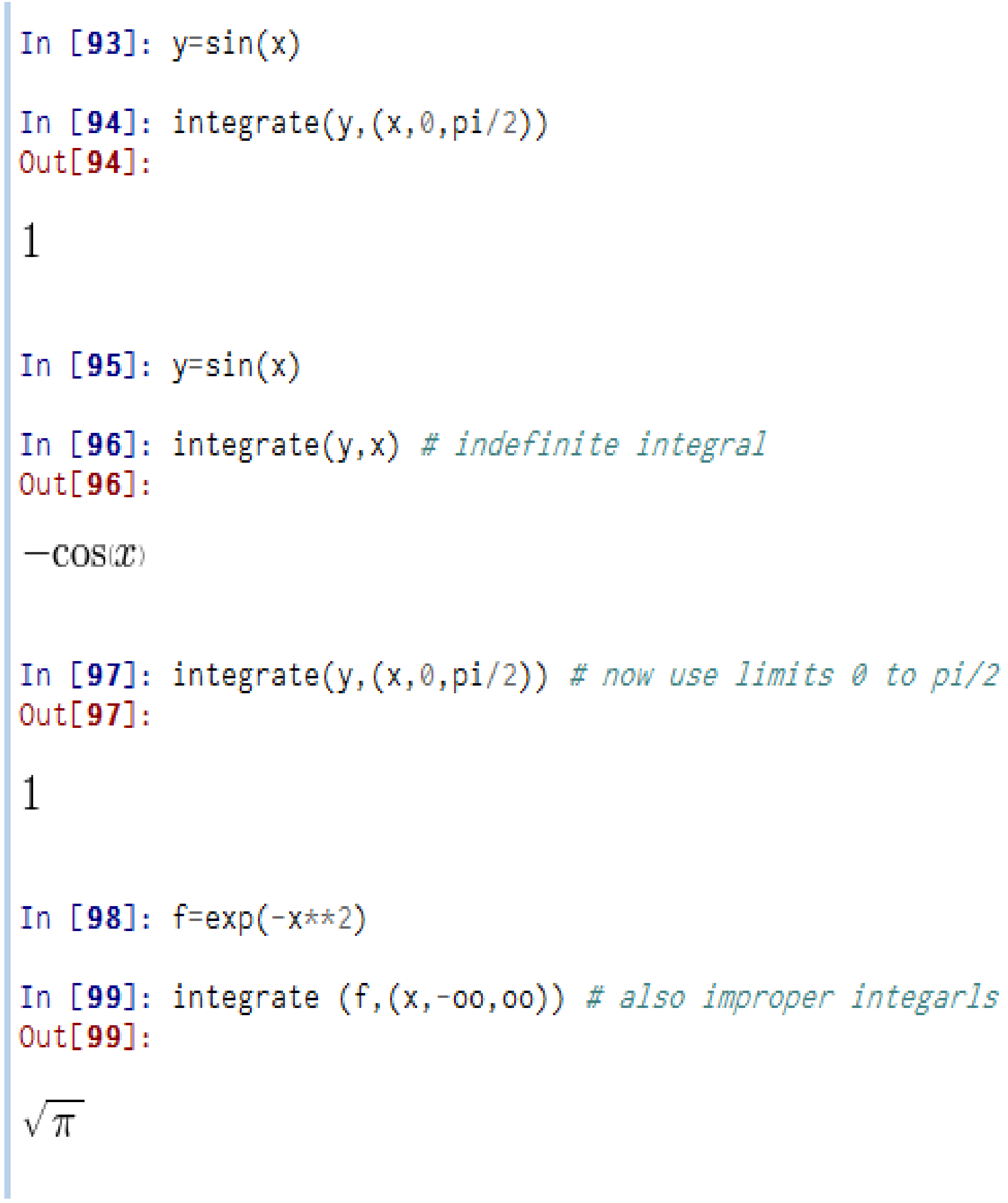
$$-xy\sin(xy) + \cos(xy)$$

```
In [78]: diff(f,y,x)
Out[78]:
```

$$-xy\sin(xy) + \cos(xy)$$

Calculus II – Integration I

Integration is done in a similar fashion using the function `integrate()`



```
In [93]: y=sin(x)

In [94]: integrate(y,(x,0,pi/2))
Out[94]:
```

1

```
In [95]: y=sin(x)

In [96]: integrate(y,x) # indefinite integral
Out[96]:
```

$-\cos(x)$

```
In [97]: integrate(y,(x,0,pi/2)) # now use limits 0 to pi/2
Out[97]:
```

1

```
In [98]: f=exp(-x**2)

In [99]: integrate (f,(x,-oo,oo)) # also improper integarls
Out[99]:
```

$\sqrt{\pi}$

Note that oo is the SymPy notation for infinity

Exercises

1. Write a program that returns the Celsius C value for a given temperature measured in Fahrenheit F. The relation between these two is given by $5(F-32)=9C$. As an example, the input 50 returns 10.

2. The time period T for a simple pendulum can be determined from

$T=2\pi\sqrt{\frac{l}{g}}$, where l is the length of string and g $(=9.81ms^{-2})$ is the constant acceleration due to gravity. Write a program to determine T for varying lengths of string.

3. Calculate the PV of £5400 in three years time given the constant risk-free interest rate is 3.4%.

4. How much is £25000 worth in five years time if the constant risk-free interest rate is 4.4% with continuous compounding.

5. Experiment with different mathematical functions from the `math` module by writing various mathematical scripts.

6. Write programs to check the following formulae by inputting n (your choice) and then computing and comparing both sides of the equation

a. $\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$ b. $\sum_{i=1}^{n} i^2=\frac{n(n+1)(2n+1)}{6}$ b. $\sum_{i=1}^{n} i^3=\frac{n^2(n+1)^2}{4}$

More exercises to follow!

There will be a further course in September!