

Python Basics

Course Outline

Day 1 - Introduction

- Goals and expectations, target audience
- Getting set up
- How to follow along
- What is Python?
- Python use cases
- Python 2 vs. Python 3
- Python vs. python
- “Coding”, “Scripting”, “Programming”
- Compiled vs. interpreted languages
- Interpreter vs. scripts
- Code editors

Day 1 - Hands-On

- Using Python on Sapelo2
- Python vs python
- Make a sample script
- Script comments
- Using the interpreter
- Executing scripts
- Python data types
 - Integers & floating point numbers
 - Strings
 - Lists
- Variables
- Python built-in functions
 - print(), len(), abs(), type(), input(), range()
- Basic Python syntax

Course Outline

Day 2 - Introduction

- Day 1 Review

Day 2 - Hands-On

- modules
- if/else statements
- for loops
- Creating and reading files
- BioPython module (working with example.fasta)

Goals and Expectations

- Understand basic Python terms & syntax
- Learn how to learn more about Python
- Learn how to write your own Python scripts
- Learn how to use Python on Sapelo2
- We have a **LOT** of information to cover
- This is a crash course!
- Be patient with yourself as you learn these new concepts

Target Audience

- Primarily Sapelo2 users, but open to all
- People brand new to coding
- Coding beginners

Getting Set Up - Sapelo2

1. Start an interactive job:
2. Search for a Python 3 module:
3. Load a Python 3 module:

```
interact
```

```
ml spider Python
```

```
ml Python/3.8.6-GCCcore-10.2.0
```

Getting Set Up - Sapelo2

Using the interpreter:

Enter the Python interpreter:

```
python
```

Exit the Python interpreter:

```
ctrl + d
```

Executing a Python script:

```
python myscript.py or chmod +x myscript.py then ./my-script.py
```

Getting Set Up - Mac

1. Open terminal
 - a. Cmd + spacebar
 - b. Type "Terminal"
 - c. Press enter
2. See if you already have Python 3 installed
 - a. `python3 --version`
 - i. If it prints out a version of Python, you're all set
 - ii. If it says "command not found"
 1. Go here: <https://www.python.org/downloads/release/python-3100/>
 2. Scroll to the bottom, download the macOS installer and install it

Getting Set Up - Mac

Once you have Python installed, in the Terminal:

Using the interpreter:

Enter the Python interpreter:

```
python3
```

Exit the Python interpreter:

```
ctrl + d
```

Executing a Python script:

```
python3 myscript.py or chmod +x myscript.py then ./my-script.py
```

Getting Set Up - Windows

1. Open PowerShell (or command prompt)
 - a. Click the start button
 - b. Search for PowerShell in the search bar
 - c. Press enter
2. See if you already have Python 3 installed
 - a. `python3 --version`
 - i. If it prints out a version of Python, you're all set
 - ii. If it says "command not found"
 1. Go here: <https://www.python.org/downloads/release/python-3100/>
 2. Scroll to the bottom, download the Windows installer and install it

Getting Set Up - Windows

Once you have Python installed, in the PowerShell/command prompt:

[Using the interpreter:](#)

Enter the Python interpreter:

```
python3
```

Exit the Python interpreter:

```
ctrl + d
```

[Executing a Python script:](#)

```
python3 myscript.py or my-script.py
```

How to Follow Along

1. Write code with me.
 - a. Inside the Python interpreter
 - b. In Python scripts
2. Ask questions!
3. Feel free to ask me to slow down or repeat anything.

What is Python?

Python is a general-purpose programming language.

- “**General purpose**” meaning it has many use-cases!
- “**Programming language**” as in we can make a computer do what we want with it.
- Python is considered a “**high level**” programming language, meaning that its syntax is generally very intuitive and readable to humans.

Python Use Cases

- Scripts/automation
- Web development
- Games
- Data science
- Bioinformatics
- Machine learning

etc...

Python 2 vs. Python 3

The integers here are the major versions of the programming language.

- **Python 2.0.1** was released on **October 16th, 2000**
- **Python 2** reached the end of full support on **January 1st, 2020**
- **Python 3.0.0** was released on **December 3rd, 2008**
- **Python 3** is still being actively developed, and has been considered the “main” version of Python long before Python 2 reached end of life, so that is what you want to learn and use.

Python 2 vs. Python 3

- Nowadays when people say “Python”, you can generally safely assume that they are referring to Python 3
- While code written in Python 2 and 3 looks similar, there are syntax differences as well as new features in Python 3, so that if you tried to execute code written in Python 3 syntax with a Python 2 interpreter, it will probably fail or not work as intended at the very least.
- For example, even simply printing out “Hello, World!” is done differently in Python 2 and 3.

If Python 2 is old news, why talk about it?

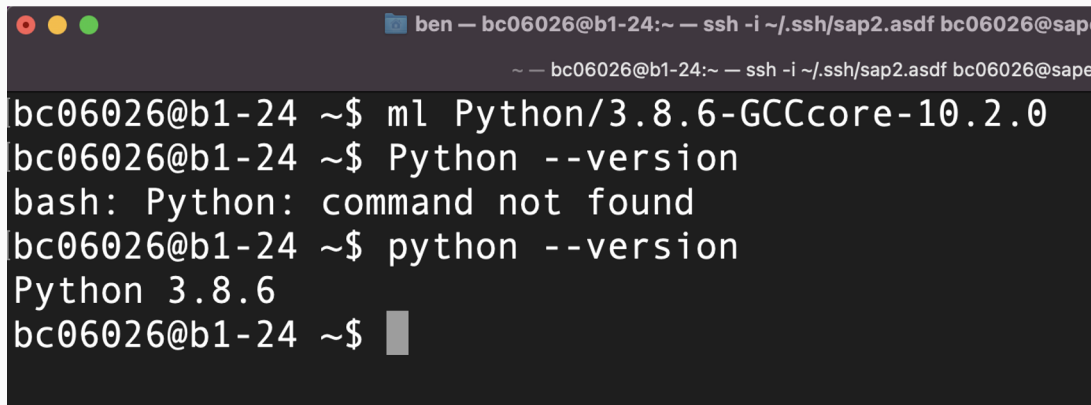
- In many Unix-like operating systems (e.g. Linux, MacOS, etc...), Python 2 comes already installed, because the operating system needs it for various things (although that is changing).
- Unless you have loaded a Python module on Sapelo2, the command `python` is probably Python 2 (which you don't want to use).
- You can check this with `python --version` in your terminal or PowerShell/Command Prompt. If that is the case you'll want to use the `python3` command to enter the Python interpreter and run scripts.

Python 2 is still around in the scientific world

- Many scientific programs out there that you may have used were written in Python 2
- Updating something originally written in Python 2 to Python 3 is no trivial task (Most researchers probably don't have time for this)
- This is fine, just be aware of what version of Python you're using, especially if you're using multiple tools at the same time in your submission script
 - `python --version`
 - `ml` (on Sapelo2, to see currently loaded modules)

Python vs python

- **Python** - Refers to the programming language when writing about the Python programming language or loading modules on Sapelo2
- **python** - refers to the command/interpreter



```
ben — bc06026@b1-24:~ — ssh -i ~/.ssh/sap2.asdf bc06026@sape
~ — bc06026@b1-24:~ — ssh -i ~/.ssh/sap2.asdf bc06026@sape
bc06026@b1-24 ~$ ml Python/3.8.6-GCCcore-10.2.0
bc06026@b1-24 ~$ Python --version
bash: Python: command not found
bc06026@b1-24 ~$ python --version
Python 3.8.6
bc06026@b1-24 ~$
```

Coding, Programming, Scripting, oh my!

Code: Instructions written for a computer to do some task(s), following some programming language's syntax.

Coding, scripting, and programming all refer to this. There is no need to be pedantic here, in my humble opinion, people, myself included, often use these terms interchangeably.

If one does make a distinction between these words, perhaps you could say:

- Coding - general term for writing code
- Programming - same as coding, maybe more likely to refer to a **compiled** programming language
- Scripting - same thing, but referring to shorter, simple programs, written in an **interpreted** programming language.

Again, no need to split hairs here.

Compiled vs. Interpreted Languages

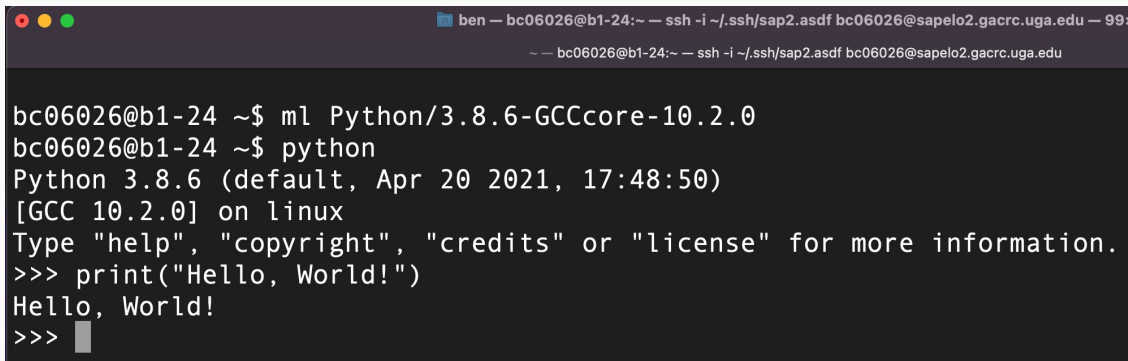
Compiled language - a programming language in which one writes the code, and then “compiles” it into a separate file (program) that the computer can understand, which then gets executed. **Examples:** C, C++, Java, Rust, Fortran

Interpreted language - a programming language in which one writes the code, and that code gets “interpreted” into what the computer can understand **at runtime**. No separate file is created that one has to execute, like with a compiled language. **Examples:** Python, Bash, Javascript, Ruby, Perl

The trade-off is that generally a compiled language could be more difficult to learn, slower to write, but will execute faster, and vice versa.

Interpreter vs. Scripts

In the context of Python, the **interpreter** is the program/command (`python`) that provides essentially a shell that executes Python commands in real-time. Example:

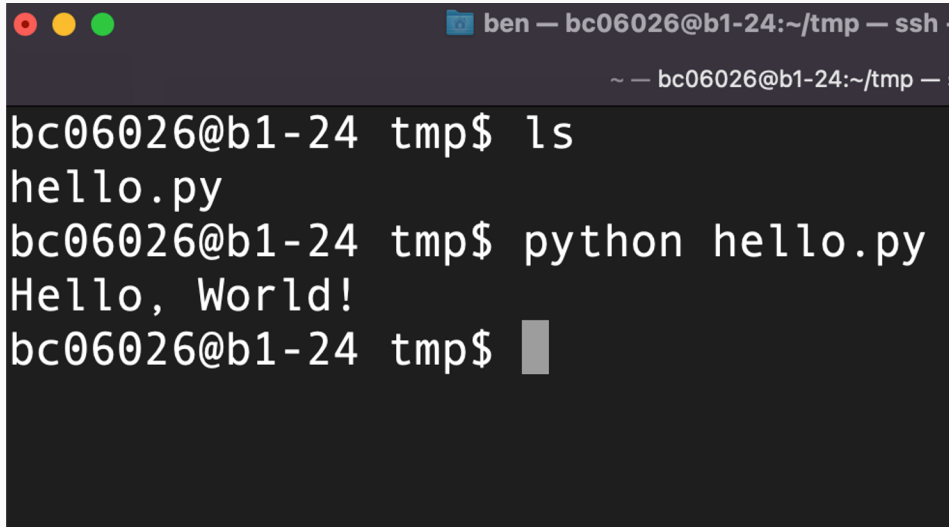


```
ben — bc06026@b1-24:~ — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 99%
bc06026@b1-24 ~$ ml Python/3.8.6-GCCcore-10.2.0
bc06026@b1-24 ~$ python
Python 3.8.6 (default, Apr 20 2021, 17:48:50)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>>
```

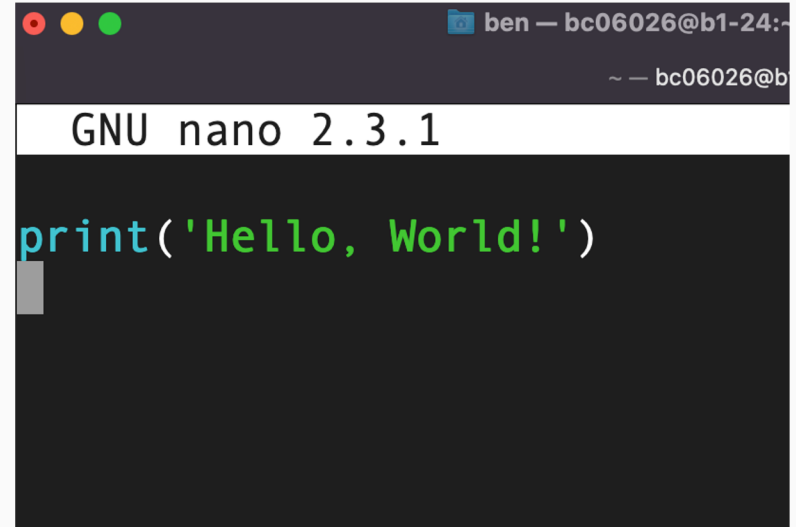
You will know that you're in the Python interpreter when your command prompt is three greater than signs (>>>). In the interpreter you can execute any valid Python code.

Interpreter vs. Scripts

A Python **script** is simply a text file that has valid Python code in it. Example:



```
ben — bc06026@b1-24:~/tmp — ssh —  
~ — bc06026@b1-24:~/tmp — s  
bc06026@b1-24 tmp$ ls  
hello.py  
bc06026@b1-24 tmp$ python hello.py  
Hello, World!  
bc06026@b1-24 tmp$ █
```



```
ben — bc06026@b1-24:~  
~ — bc06026@b  
GNU nano 2.3.1  
print('Hello, World!')  
█
```

Interpreter vs. Scripts

So when would one want to write a Python script versus executing Python code within the interpreter?

- The **interpreter** is going to generally be for testing things out, debugging, quick usage. The key concept is that the interpreter executes one line of Python code at a time, in real time.
- A Python **script** could be for something you intend to run more than once, or something that you want to share with others. With a script you have an actual file that you can repeatedly execute, share, and tweak as needed. If you wanted to execute your own Python code in a submission script on Sapelo2, you would need to write a Python script to be executed within your job.

Do **not** run Python code whether via the interpreter or a script on the login/submit nodes!!!
Only do this within a job, be that interactive (`interact`) or a batch job (`sbatch sub.sh`)

Code Editors

- Code editors can be thought of as a text editor program that you can install on your own computer, designed specifically for coding.
- There are many free code editors out there that provide some nice quality of life features such as syntax color highlighting, syntax error highlighting, auto-completion, etc..
- You do not have to write code in a code editor!
- Some popular examples: VS Code, Atom, PyCharm, Sublime
- If you use a code editor, which one you use is largely a matter of personal preference

Let's execute our first script!

On Sapelo2:

If you haven't already:

```
interact  
ml Python/3.8.6-GCCcore-10.2.0
```

On Mac:

1. Open your terminal
2. `cd` to where you will save your Python scripts
3. Get ready to use a code editor or nano in your terminal to write your first script

On Windows:

1. Open PowerShell or Command Prompt
2. `cd` to where you will save your Python scripts
3. Open a code editor, or if you don't have one installed, notepad to write your first script

Let's execute our first script!

- | | |
|-------------------------------|----------------------------------|
| 1. cd to your /scratch dir: | <code>cd /scratch/\$USER</code> |
| 2. Make a subdirectory: | <code>mkdir py-training</code> |
| 3. cd to that directory: | <code>cd py-training</code> |
| 4. Create a script with nano: | <code>nano hello.py</code> |
| 5. Save and exit nano: | <code>ctrl + x , y, enter</code> |
| 6. Execute the script! | <code>python hello.py</code> |



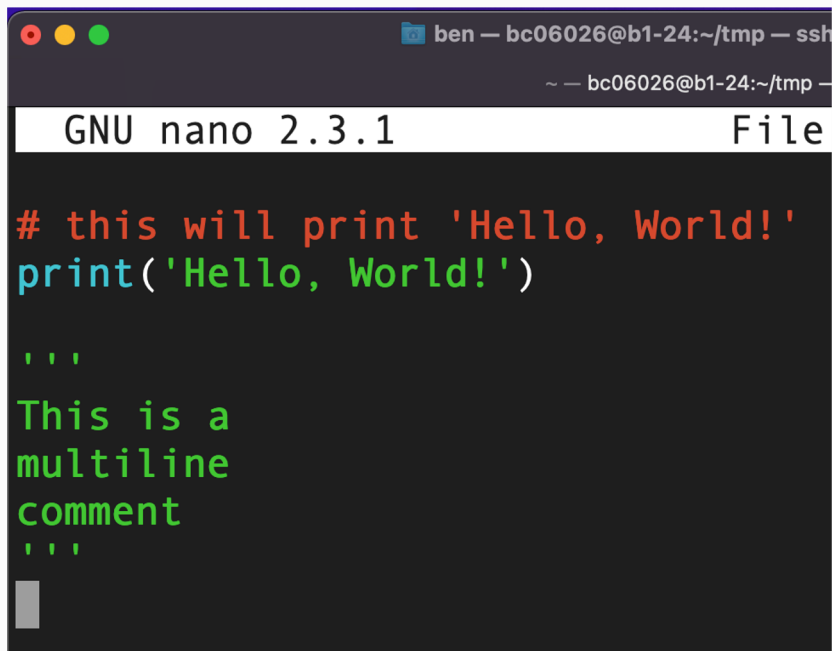
```
ben — bc06026@c4-16:/scratch/bc06026/py-training — s
GNU nano 2.3. File: hello.py
print('Hello, World!')
```

```
ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/sap2.asdf bc06026...
bc06026@c4-16 py-training$ python hello.py
Hello, World!
bc06026@c4-16 py-training$
```

Comments

In Python (and programming languages in general), there is a convention to write **comments** in one's code. Comments are for informational purposes and are **ignored by Python when code is executed**. You can create a comment with a `#` sign or a multi-line comment enclosed in triple quotation marks.

These are simple examples, but you generally want to use comments to easily explain what your code does not.



```
ben — bc06026@b1-24:~/tmp — ssh
~ — bc06026@b1-24:~/tmp —
GNU nano 2.3.1 File
# this will print 'Hello, World!'
print('Hello, World!')

'''
This is a
multiline
comment
'''
```


More on Comments

You will also see and hear people talk about “**commenting (something) out**”. This simply means to put `#` signs or triple quotes in front of/around code **temporarily**, so that it will not execute. This is typically done while testing and developing code. It can be useful to temporarily disable some portion of your code, without losing it.

```
ben — bc06026@b1-24:~/tmp — ssh -i ~ — bc06026@b1-24:~/tmp — ssh  
GNU nano 2.3.1  
print('Hello, World!')  
print(3.14 ** 2)
```



```
ben — bc06026@b1-24:~/tmp — ssh -i ~ — bc06026@b1-24:~/tmp — ssh  
bc06026@b1-24 tmp$ python hello.py  
Hello, World!  
9.8596  
bc06026@b1-24 tmp$
```

```
ben — bc06026@b1-24:~/tmp — ssh -i ~ — bc06026@b1-24:~/tmp — ssh  
GNU nano 2.3.1  
print('Hello, World!')  
# print(3.14 ** 2)
```



```
ben — bc06026@b1-24:~/tmp — ssh -i ~ — bc06026@b1-24:~/tmp — ssh  
bc06026@b1-24 tmp$ python hello.py  
Hello, World!  
bc06026@b1-24 tmp$
```

More on executing scripts...

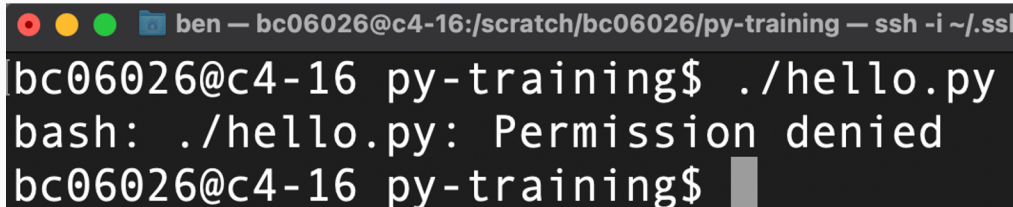
The two ways you'll typically see a Python script executed are the way we just did it:

```
python myscript.py
```

Or alternatively:

```
./myscript.py
```

The end result will be the same, in that both ways execute the script, but the latter requires a little bit of setup and explanation. If you try to execute a script with `./` without the minor setup, you may see something like this:



```
ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/
bc06026@c4-16 py-training$ ./hello.py
bash: ./hello.py: Permission denied
bc06026@c4-16 py-training$
```

More on executing scripts...

Permission denied? What's going on there?

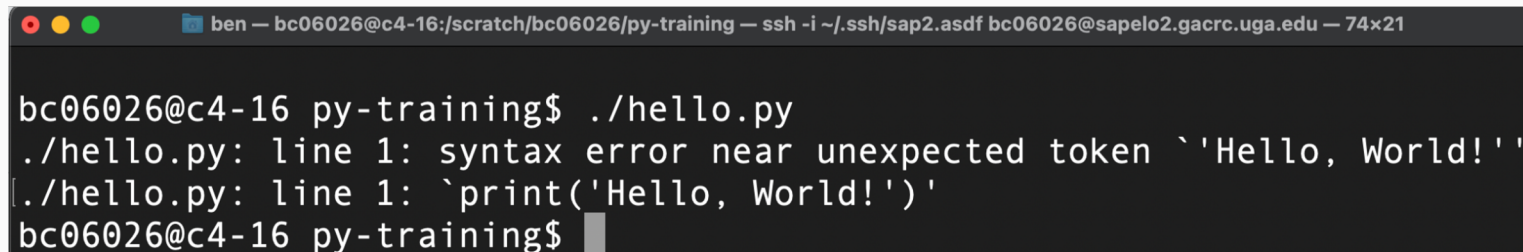
```
ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 68x21
bc06026@c4-16 py-training$ ls -l hello.py
-rw-r--r-- 1 bc06026 gacrc-instruction 23 Dec  2 10:43 hello.py
bc06026@c4-16 py-training$
```

The execute permission is not set!

```
ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 68x21
bc06026@c4-16 py-training$ chmod +x hello.py
bc06026@c4-16 py-training$ ls -l hello.py
-rwxr-xr-x 1 bc06026 gacrc-instruction 23 Dec  2 10:43 hello.py
bc06026@c4-16 py-training$
```

More on executing scripts...

Will it work now?

A terminal window with a dark background and light text. The title bar at the top shows a red, yellow, and green window control icon, followed by the text "ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 74x21". The terminal content shows a user prompt "bc06026@c4-16 py-training\$" followed by the command "./hello.py". The next line shows an error message: "./hello.py: line 1: syntax error near unexpected token `'Hello, World!'"", followed by another line: "./hello.py: line 1: `print('Hello, World!')'". The prompt "bc06026@c4-16 py-training\$" appears again at the end of the output, with a cursor block following it.

```
ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 74x21

bc06026@c4-16 py-training$ ./hello.py
./hello.py: line 1: syntax error near unexpected token `'Hello, World!'"
./hello.py: line 1: `print('Hello, World!')'
bc06026@c4-16 py-training$
```

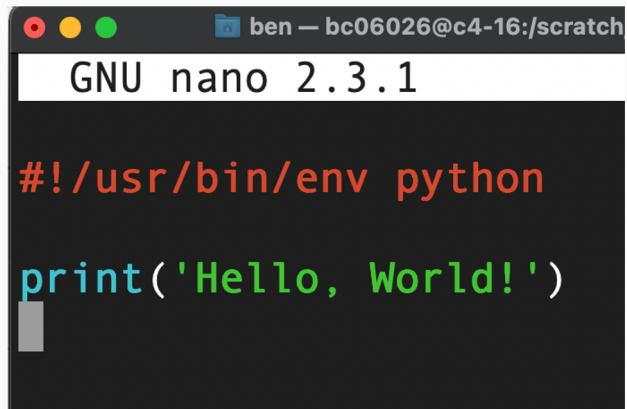
It says we have a “syntax error”, as if our code is wrong. The problem is that the computer does not know what program to use to execute our script. Because we didn’t specify that, it defaulted to Bash (Linux command line), instead of what we want it to use, Python.

The “shebang” #!

- To fix this, we need to add to the top of our script something called a **shebang**. A shebang is a directive that tells the computer reading the script what program to use to execute it.
- **Linux does not care that we put a “.py” at the end of the script name.** That is meaningless as far as the computer is concerned and is just a naming convention for humans.
- A shebang starts with `#!` followed by the path to the program to be used to execute a script.
- You may often see shebang’s that look like: `#!/usr/bin/python`. The issue with that is that it assumes the location of your Python interpreter. Instead, you can use the “env” program to find in your PATH which Python interpreter to use (this is very applicable when using modules on an HPC cluster).
- For example on Sapelo2, after loading a Python module: `#!/usr/bin/env python`
- Note that this does not apply to Windows. In Windows you are restricted in how you can name your files, because it uses its registry to determine how to execute things like this.

Implementing the shebang

1. Add the shebang to the top of your script
2. Execute it!

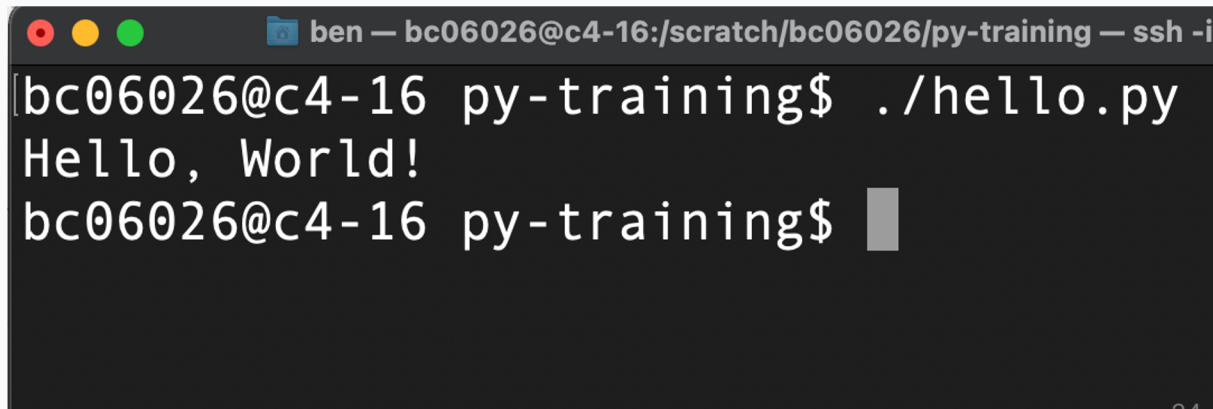


A terminal window titled "ben — bc06026@c4-16:/scratch" showing the GNU nano 2.3.1 text editor. The editor contains a Python script with a shebang line and a print statement.

```
GNU nano 2.3.1

#!/usr/bin/env python

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



A terminal window titled "ben — bc06026@c4-16:/scratch/bc06026/py-training — ssh -i" showing the execution of the script. The prompt is "bc06026@c4-16 py-training\$". The command "./hello.py" has been entered, and the output "Hello, World!" is displayed. The prompt is now "bc06026@c4-16 py-training\$".

```
bc06026@c4-16 py-training$ ./hello.py
Hello, World!
bc06026@c4-16 py-training$
```

Python Data Types

In Python, everything is an **object**. An object is an **instance** of a **class**, also called a “**type**”.

For example, in the Python interpreter, we can use the built-in `type()` function to see what kind of object something is:

In these three examples, we see:

- the object **1** belongs to the **int** class
- the object **1.0** belongs to the **float** class
- the object **'Hello,World!'** belongs to the **str** class

In plain English, we could say, “*1 is an integer, 1.0 is a floating point number, and 'Hello, World!' is a string.*”

A terminal window with a dark background and light text. The title bar shows a red, yellow, and green window control icon, followed by the text "ben — bc06026@c4-16:/scratch/bc06026/py-training". The terminal contains three lines of Python code and their corresponding outputs:

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
>>> █
```

Python Data Types - int, float

<https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

- **int** and **float** are two of Python's numeric types
- There are many operators that can be used with int and float objects

$x + y$	Sum of x and y
$x - y$	Difference of x and y
$x * y$	Product of x and y
x / y	Quotient of x and y
$x // y$	Floored quotient of x and y
$x ** y$	x to the power y

Python Data Types - str

<https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

- The str (string) type in Python is any sequence of characters enclosed by quotation marks, either single or double.

Examples:

"Hello, world!"

"x"

'y'

"CCAGCCGGACTTCAGGCCTGCCATCC
AGTTCCCGCGAAGCTGGTCTTCAGCC"

'42'

"3.1415926535"

"!@#\$%^&*()"

Variables

A variable is a symbolic name for a particular object. If you have some value you want to store and use throughout a program, you can define a variable to equal this value, and reference the variable name rather than the object whenever the value is used. This makes code much easier to read and maintain.

To create a variable in Python, write the variable name followed by an equals sign, and then the value you wish to assign to the variable.

Examples:

```
x = 3.1415926535
```

```
name = "John"
```

```
year = 2022
```

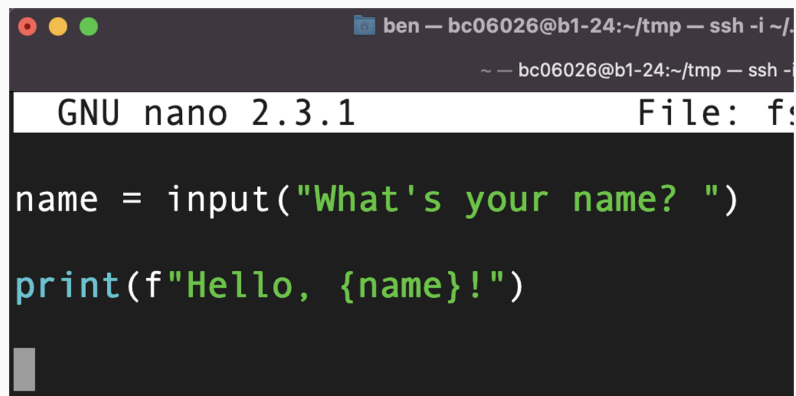
```
input_files = ['input1.dat', 'input2.dat', 'input3.dat']
```

Variables

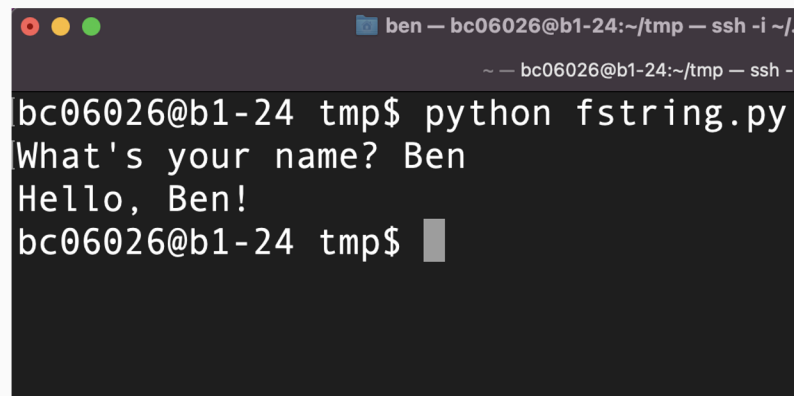
- Variable names are case sensitive (and typically written in lowercase).
- Variable names may start with an underscore or a letter, and may contain numbers.
- While one may use a single character such as “x” as a variable name, in most cases it is advisable to use a variable name that is descriptive enough to give a reader of the code an idea of what’s going on.
- Python is a **dynamically-typed** language which means that you do not have to specify what **type** of object a variable represents. For example, given the variable assignments `x = 0` and `y = 3.14`, Python infers on its own that `x` is an `int` and `y` is a `float`.

Formatted Strings

One fun way we can use variables is to insert them into strings using a convention called an “f string”. To make an f string, put a lowercase “f” immediately before the quotation marks and insert a variable name into the string surrounded by curly braces.



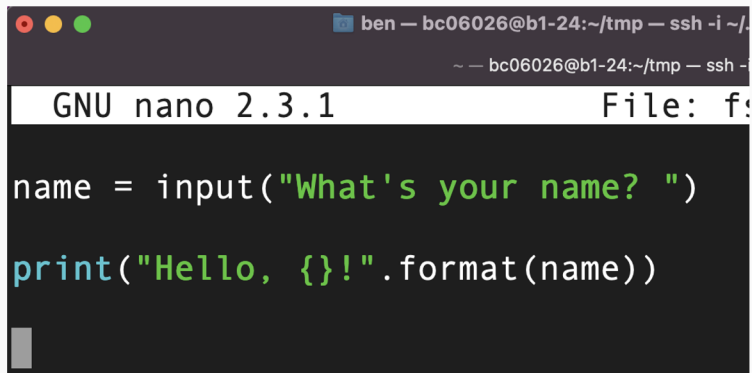
```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/bc06026@b1-24
GNU nano 2.3.1 File: fstring.py
name = input("What's your name? ")
print(f"Hello, {name}!")
```



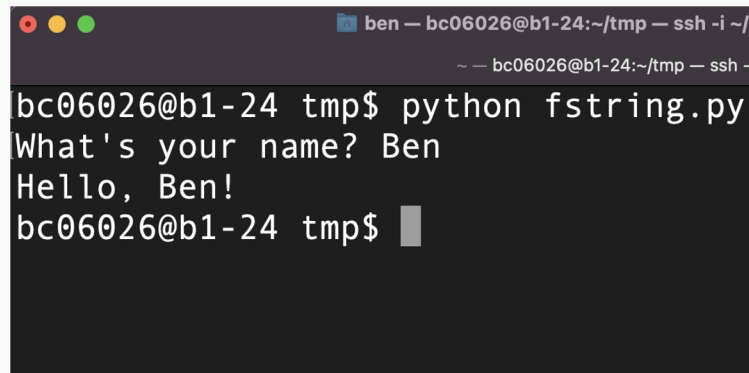
```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/bc06026@b1-24
bc06026@b1-24 tmp$ python fstring.py
What's your name? Ben
Hello, Ben!
bc06026@b1-24 tmp$
```

Formatted Strings

f strings were introduced in Python 3.6. The old way to insert variables into strings (which still works in Python ≥ 3.6) is to use the format method of a string as follows:



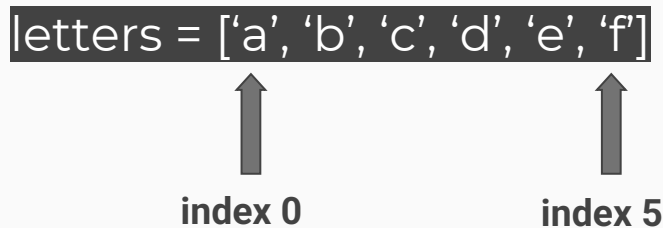
```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/bc06026@b1-24-ssh-key.pem  
GNU nano 2.3.1 File: fstring.py  
name = input("What's your name? ")  
print("Hello, {}!".format(name))  
█
```



```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/bc06026@b1-24-ssh-key.pem  
bc06026@b1-24 tmp$ python fstring.py  
What's your name? Ben  
Hello, Ben!  
bc06026@b1-24 tmp$ █
```

Python Data types - list

- A list is a mutable, ordered, comma-separated sequence of objects.
- Each object in a list is called an **element**
- Elements in a list do not have to be of the same type
- An element's position in a list is called its **index**, which is an integer value that starts counting from 0
- Lists are created using square braces



The diagram shows a Python list named 'letters' containing six string elements: 'a', 'b', 'c', 'd', 'e', and 'f'. Below the list, two upward-pointing arrows indicate the indexing of the first and last elements. The first arrow points to the element 'a' and is labeled 'index 0'. The second arrow points to the element 'f' and is labeled 'index 5'. This illustrates that list indices start at 0 and increase by 1 for each subsequent element.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f']
```

index 0 index 5

List indexing

To access an element by its index, we can put at the end of a list additional square braces which contain the element we want to access.

This is called **indexing** a list.

Given the list:

```
letters = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
print(letters[0])
```

 would print "a"

```
print(letters[3])
```

 would print "d"

```
print(letters[-1])
```

 would print "f"

```
print(letters[6])
```

 would give an
"IndexError: list index out of range" error

List slicing

We can access multiple list elements by **slicing** a list. This expands upon the indexing notation we saw in the previous slide. The syntax is:

```
mylist[start:end:step]
```

start: the index from which you start (default: 0)

end: the index up to which you end (default: end of list)

step: the increment at which you traverse through list elements to create your sliced list (default: 1)

List slicing examples

Given the list:

```
my_list = ['Python', 'is', 'lots', 'of', 'fun!']
```

```
print(my_list[2:]) --> ['lots', 'of', 'fun']
```

```
print(my_list[2:4]) --> ['lots', 'of']
```

```
print(my_list[:2]) --> ['Python', 'is']
```

```
print(my_list[::-2]) --> ['Python', 'lots', 'fun']
```

```
print(my_list[::-1]) --> ['fun', 'of', 'lots', 'is', 'Python']
```

Note:

- If used, the “end” index spot goes up to, but **not including** the provided index
- if you don't include the extra colon + the “step”, it defaults to 1 (traversing the list one element at a time).

Built-in Functions

- A **function** in Python is some code referenced by a name. It is as if a variable is referencing one or more lines of code that are designed to perform some task.
- You can create your own functions, but Python has many useful **built-in functions**.
- We have already seen two built-in functions `print()` in our first script and `type()` when introducing objects.
- Functions have opening and closing parentheses appended to their names.
- Anything put in the parentheses is called an **argument** to the function. Which means it is given to the function as input for whatever the function does.

Built-in Functions Examples

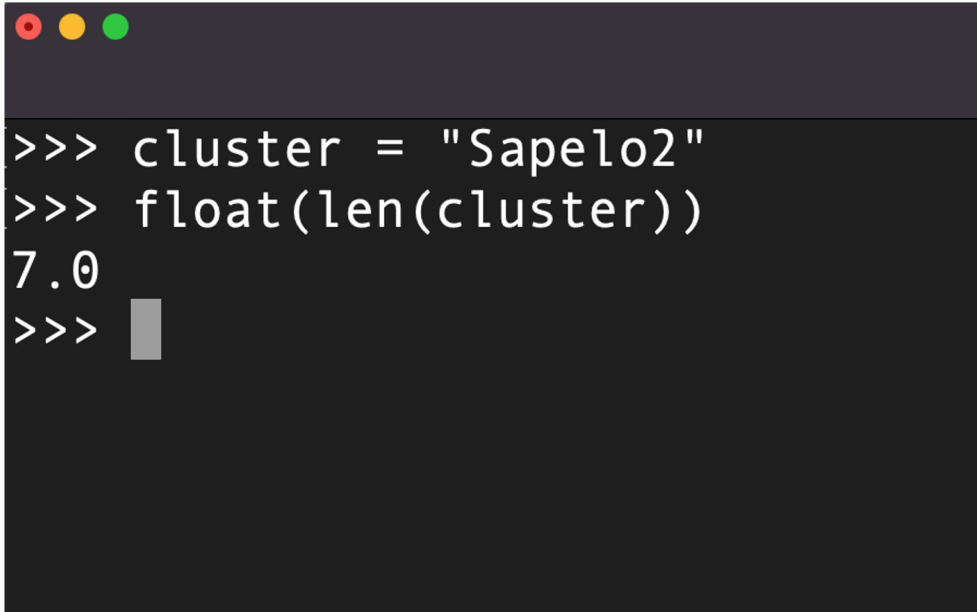
<code>print()</code>	prints out its argument
<code>type()</code>	returns what type of object something is
<code>len()</code>	returns the length of an object (e.g., the length of a str)
<code>abs()</code>	returns the absolute value of an object
<code>int()</code>	returns an object converted to an int
<code>float()</code>	returns an object converted to a float
<code>str()</code>	returns an object converted to an str
<code>input()</code>	gets input from the user

Nested Functions

Functions can be **nested**. This means that you can pass the **return value** of a function to another function.

In this example, the return value of `len(cluster)` (7), is being passed to the built-in `float` function, which then returns 7 converted to a float.

Pay attention to the closing and opening parenthesis when you have nested functions like this.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a Python interactive session where a string 'Sapelo2' is assigned to a variable named 'cluster', and then the length of 'cluster' is passed to the 'float' function, resulting in the output '7.0'.

```
>>> cluster = "Sapelo2"
>>> float(len(cluster))
7.0
>>> 
```

Modules

<https://docs.python.org/3/py-modindex.html>

A Python module is simply a file ending in .py with Python code in it, that can be “imported” into your scripts (not to be confused with an HPC cluster software module). Without going into too much detail, you can think of Python modules as 3rd party libraries. Python has many modules, both built in and third party, that you can use to extend the functionality of your code (don’t reinvent the wheel).

os	miscellaneous operating system interfaces
time	time access and conversions
re	regular expression operations
threading	thread-based parallelism
multiprocessing	process-based parallelism

Modules

After **importing** a Python module into your script or interpreter, you can access any functions it provides. In this example, I use the `os` module to check if a file exists in the current directory.

```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 80x24
~ — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu
bc06026@b1-24 tmp$ ls
name.py
bc06026@b1-24 tmp$ python
Python 3.9.5 (default, Feb  1 2022, 14:49:35)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.path.exists('name.py')
True
>>> 
```

if/else statements

<https://docs.python.org/3/library/stdtypes.html#comparisons>

if/else statements allow us to introduce flowchart-esque functionality to our Python code.

if/else statements check a given condition. In this screenshot, it is checking if the length of name is greater than 4.

If the condition evaluates as True, any code indented under “if” will be executed, and the “else” statement will be skipped.

If the condition evaluates as False and an “else” statement follows the “if” statement, then the indented code under the “if” statement will **not** be executed, and any code indented under “else” will be executed.

Given the code in this screenshot, what would be printed if `name` were equal to "Matthew"? What about "Jane"?

```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@b1-24:~/tmp
```

```
GNU nano 2.3.1 File: ifelse.py

name = input("what's your name? ")

if len(name) > 4:
    print("your name has more than 4 letters")
else:
    print("your name has 4 or less letters")
```

Python and Indentation

One aspect of Python that is very unique is its designation of blocks of code via indentation rather than curly braces. For example, Python knows what to execute for if/else statements by virtue of what code is indented, whereas most other programming languages would designate that by surrounding code with { }.

It is a subject of great debate as to whether it is “better” to indent your code with tabs or spaces. Whichever you prefer, you must be consistent.

If part of your code is indented using tabs and other parts spaces, Python will complain (“**TabError: inconsistent use of tabs and spaces in indentation**”). This is something you may encounter when copying and pasting code from the internet into your own code.

If your code is indented with an inconsistent amount of spaces, Python will complain (“**IndentationError: unexpected indent**”).

if/else statements

You do not have to have an “else” statement after your “if” statement if you don’t want/need to. In such a case, the “if” statement gets checked, and if it evaluates as False, the indented code is simply skipped.

```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu — 81x24
~ — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026@sapelo2.gacrc.uga.edu

GNU nano 2.3.1                               File: ifelse.py

training = "Python"

if training == "Python":
    print("Make sure to indent properly!")

print("This will print regardless of the result of the above if statement.")

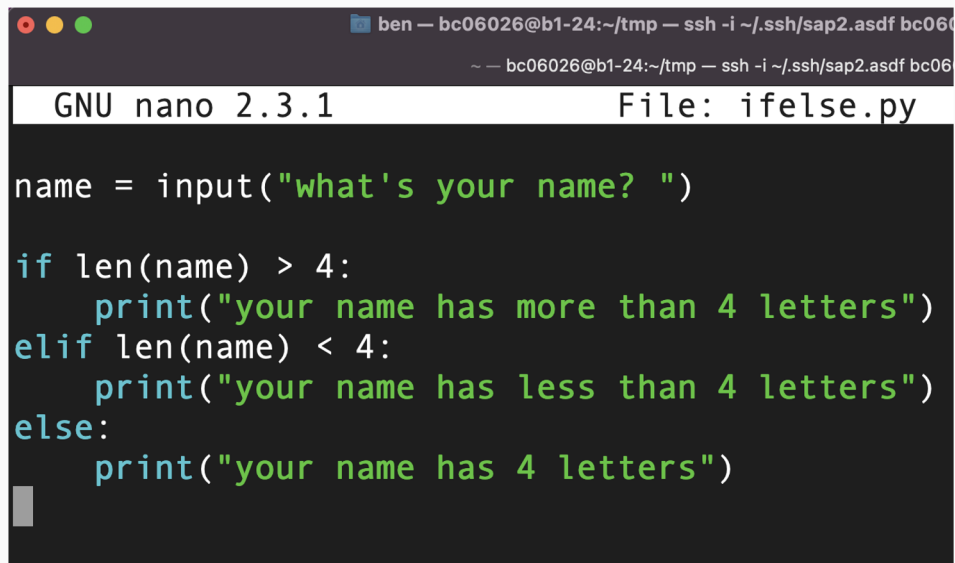
[]
```

* Note the difference between the **assignment** operator = and the **comparison** operator ==

if/else statements

You can also insert “elif” statements after an “if” statement (and before an “else” statement if there is one). “elif” stands for “else if”. The purpose of an “elif” statement is to provide subsequent condition(s) to be checked.

You can have more than one “elif” statement, just keep in mind that they get checked in the order that they’re listed, and no following conditions get checked as soon as a condition evaluates to true.



```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc060
~ — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc060
GNU nano 2.3.1 File: ifelse.py

name = input("what's your name? ")

if len(name) > 4:
    print("your name has more than 4 letters")
elif len(name) < 4:
    print("your name has less than 4 letters")
else:
    print("your name has 4 letters")
```

if/else statements

There is no limit to how many statements you can have under an if/elif/else statement. Just make sure you indent properly.



```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026
~ — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asdf bc06026
GNU nano 2.3.1 File: ifelse.py

name = input("what's your name? ")

if len(name) > 4:
    print("your name has more than 4 letters")
elif len(name) < 4:
    print("your name has less than 4 letters")
else:
    print("your name has 4 letters")
    print("this is the else statement")
```

for loops

For loops allow us to **iterate** through objects/data. For example, imagine you have a **list** of objects, and you want to perform some operation(s) on all of them.

```
ben — b
GNU nano 2.3.1
nums = [1,2,3,4,5]
for x in nums:
    print(x**2)
```



```
ben — bc06026@b1-24:~/tmp —
~ — bc06026@b1-24:~/tr
bc06026@b1-24 tmp$ python for.py
1
4
9
16
25
bc06026@b1-24 tmp$
```

In the above example, “x” is considered to be a variable that only exists within the scope of the for statement. We could call that variable anything we want to.

for loops

If you have a reason to do something x number of times, you can use the built-in range function as the thing that is being iterated through in a for loop.

```
ben — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asd
~ — bc06026@b1-24:~/tmp — ssh -i ~/.ssh/sap2.asd
GNU nano 2.3.1                               File: for.py
for num in range(3):
    print("This will be printed 3 times")
```



```
ben — bc06026@b1-24:~/tmp —
~ — bc06026@b1-24:~/tmp —
bc06026@b1-24 tmp$ python for.py
This will be printed 3 times
This will be printed 3 times
This will be printed 3 times
bc06026@b1-24 tmp$
```

for loops

You can also nest if/else statements inside for loops.

```
ben — bc06026@b1-24:~/tmp —  
~ — bc06026@b1-24:~/tr  
GNU nano 2.3.1 F  
  
for num in range(10):  
    if num % 2 == 0:  
        print(f'{num} is even')
```



```
ben — bc06026@b1-24:~/tmp —  
~ — bc06026@b1-24:~/tm  
bc06026@b1-24 tmp$ python for.py  
0 is even  
2 is even  
4 is even  
6 is even  
8 is even  
bc06026@b1-24 tmp$
```

* Note that the range function starts at 0 and goes up to but does not include the argument provided to it. e.g., range(5) would include 0, 1, 2, 3, 4.

Working with Files

One of the most tangible and fun ways we can use Python is to interact with files on a computer. We can create and append to files using the built-in `open()` function.

```
ben — bc06026@b1-24:~  
~ — bc06026@b1-24:~  
GNU nano 2.3.1 File  
  
with open("file.txt", "w") as f:  
    f.write("Hello from Python!\n")  
█
```



```
ben — bc06026@b1-24:~/tmp  
~ — bc06026@b1-24:~/tmp  
bc06026@b1-24 tmp$ ls  
new_file.py  
bc06026@b1-24 tmp$ python new_file.py  
bc06026@b1-24 tmp$ ls  
file.txt new_file.py  
bc06026@b1-24 tmp$ cat file.txt  
Hello from Python!  
bc06026@b1-24 tmp$ █
```

“with” is a convention we can use to ensure Python appropriately closes the given file, without you the programmer having to manually ensure that that happens. In the above example, “w” means to write to the given file and create it if it doesn’t exist. The “f” variable only exists within the scope of that with statement, and it represents the given file. Note that the above file gets created in the current directory, because my given path was only the name of the

Working with Files

Other modes in which you can interact with a file via the `open()` function:

"a"	Append - will append to the end of the file, or create if it doesn't exist
"w"	Write - will overwrite any existing content, or create if it doesn't exist
"x"	Create - will create a file, returns an error if the file exists
"r"	Read - will open a file to be read