OAK RIDGE National Laboratory

Intro to Python Programming

Michael Sandoval

HPC Engineer - User Assistance Group

Oak Ridge Leadership Computing Facility (OLCF)

Oak Ridge National Laboratory (ORNL)

December 15, 2022

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

Crash Course Material

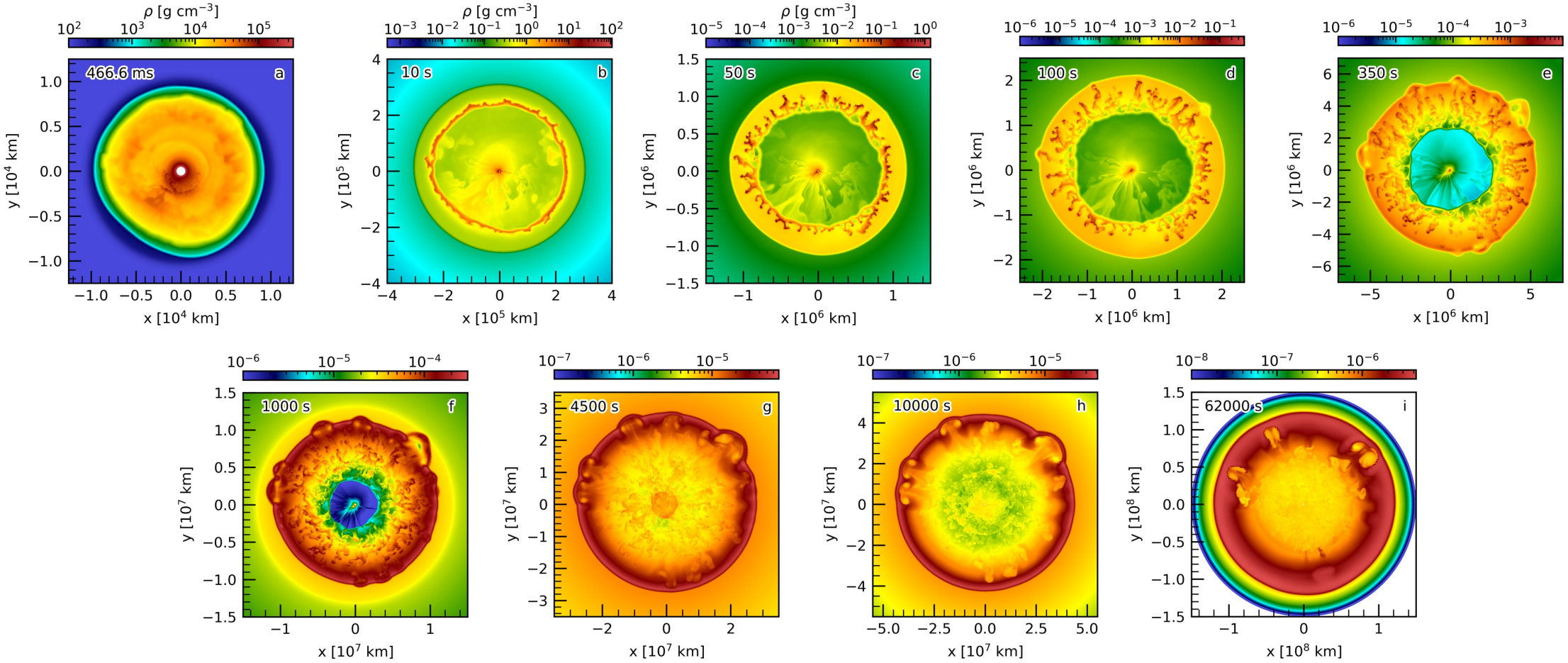
- SSH into the OpenDTNs / clone the repo if you haven't already
- cd ~/foundational_hpc_skills/intro_to_python
- Following along interactively during presentation is optional
- Let's see how far we can get!
- Walkthrough on github repo (use the README in browser)
  - Step-by-step tutorial
  - Premade scripts
  - Bonus challenges "homework" at the end

https://github.com/olcf/foundational_hpc_skills/tree/master/intro_to_python

What is Python?

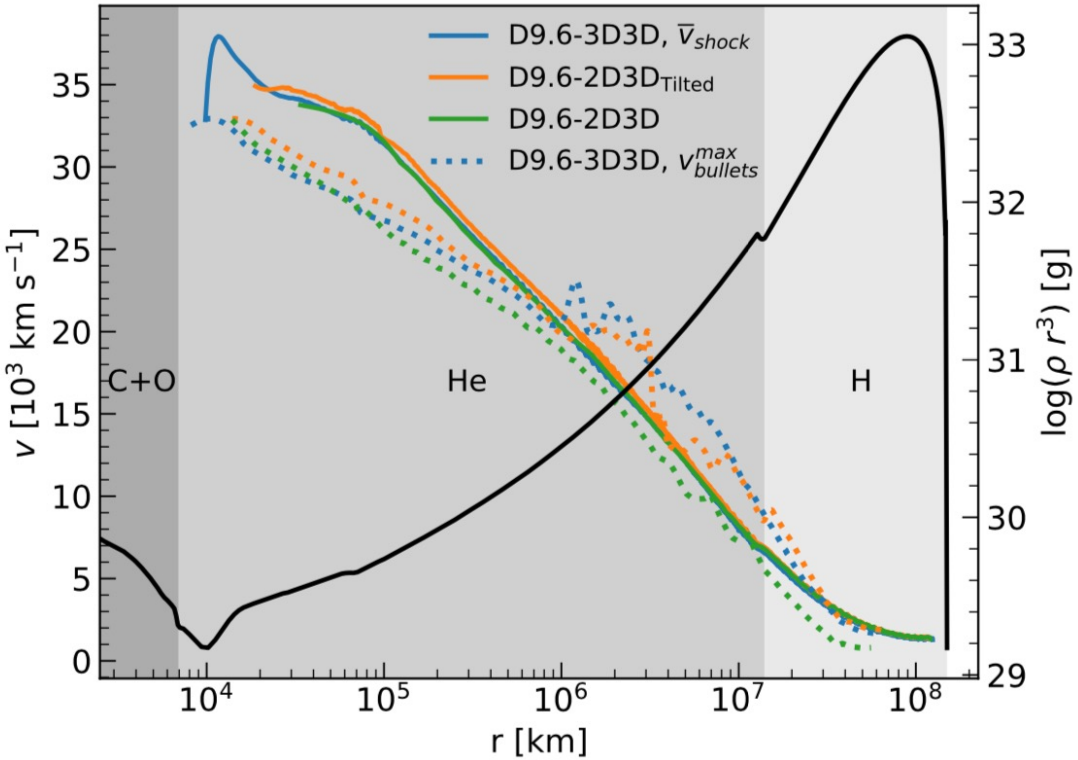
- Interpreted language (vs. compiled language)
  - Interpreted (Python, JavaScript): translates line-by-line
  - Compiled (C/C++, Fortran): send your code off to be fully translated
- Beginner friendly
- In HPC:
  - Mainly used for analysis (not speedy enough for traditional HPC)
  - Gaining momentum via: Machine Learning, Quantum, CuPy (GPUs)
- So it doesn't do any of the "fun" HPC stuff? → Wrong!
  - With HPC comes big data → Python is good at visualizing big data (pretty pictures!)

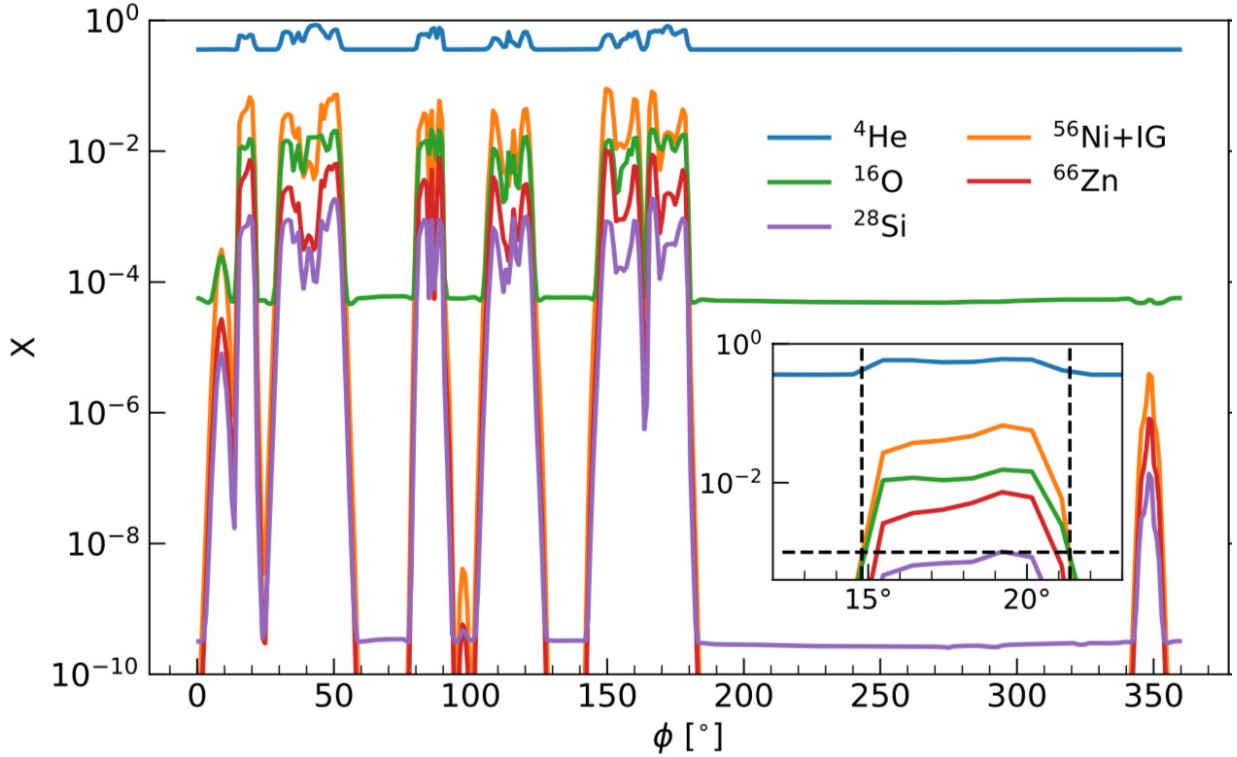
Slice Plots



Sandoval et al 2021

Line Plots





Sandoval et al 2021

Animations

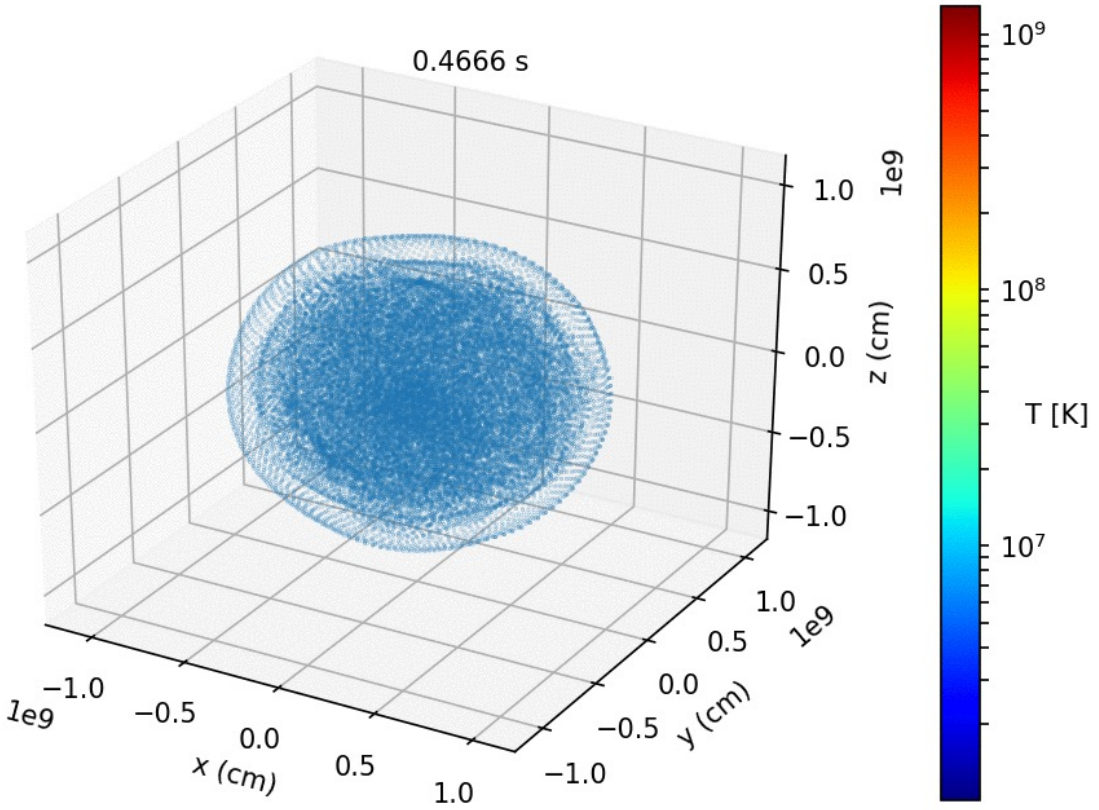
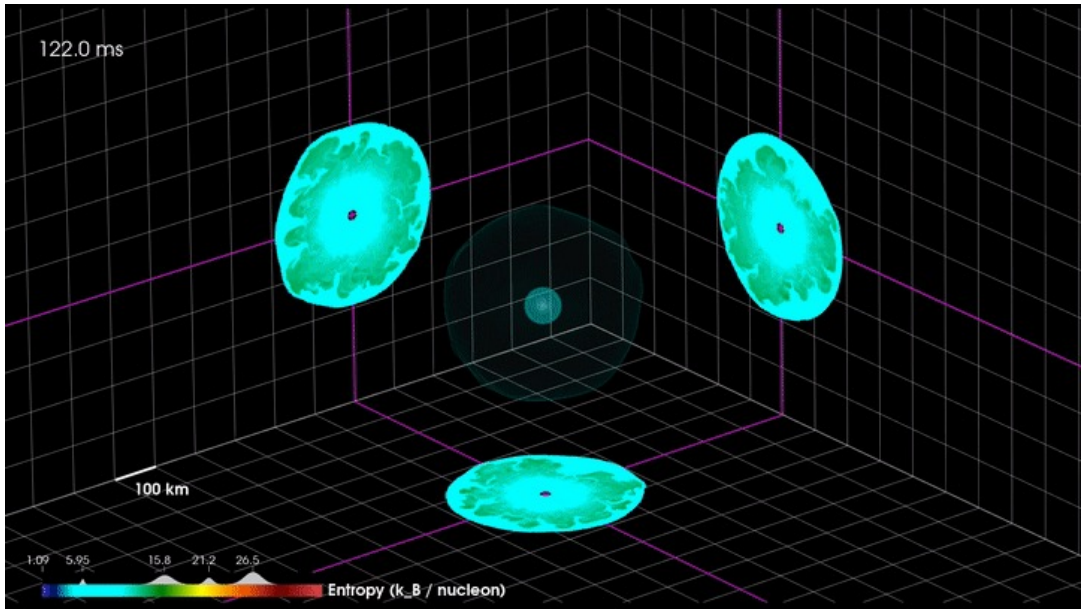
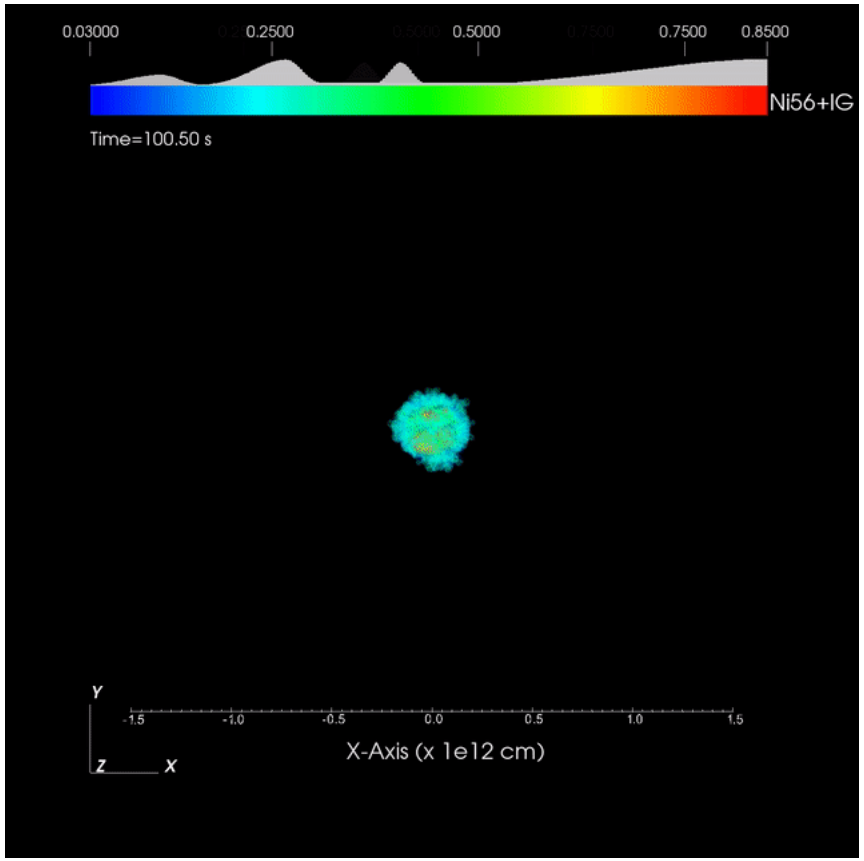
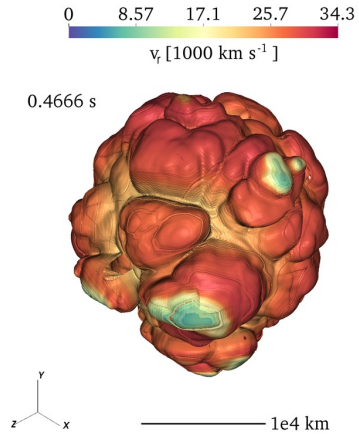


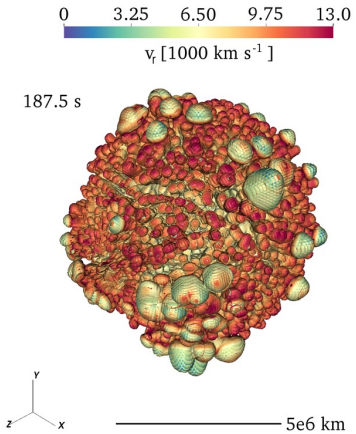
Image credit: Michael Sandoval

3D Modeling – VisIt (Python based)









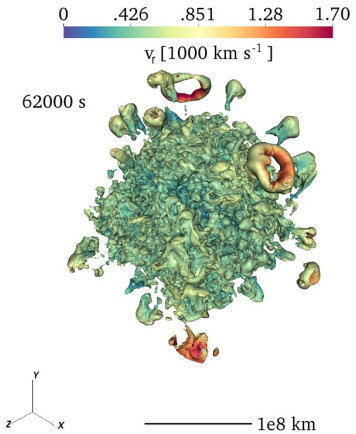


Image credit: Michael Sandoval

3D Modeling - ParaView (Python based)

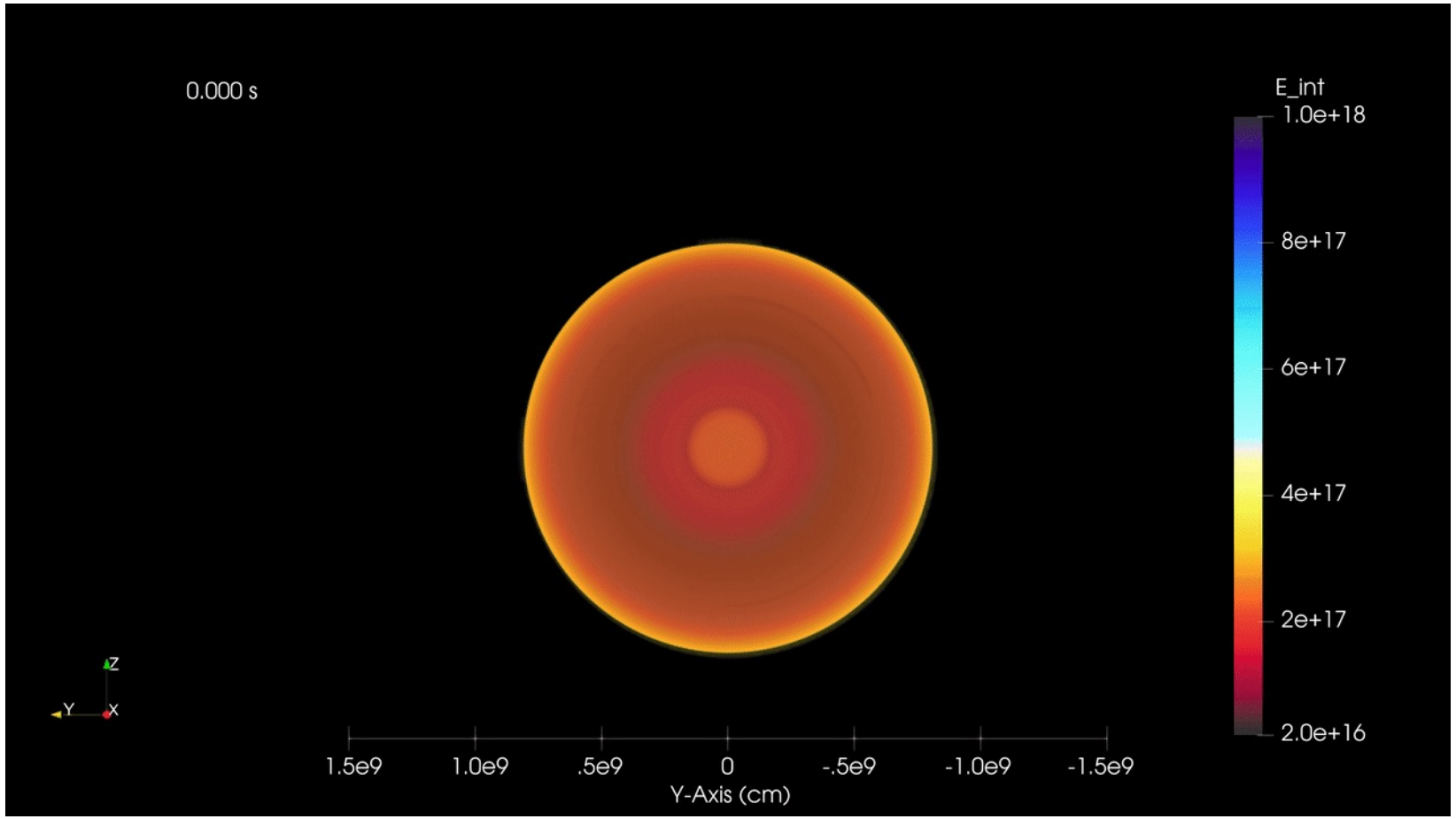


Image credit: Michael Sandoval

3D Modeling – Blender (Python based)

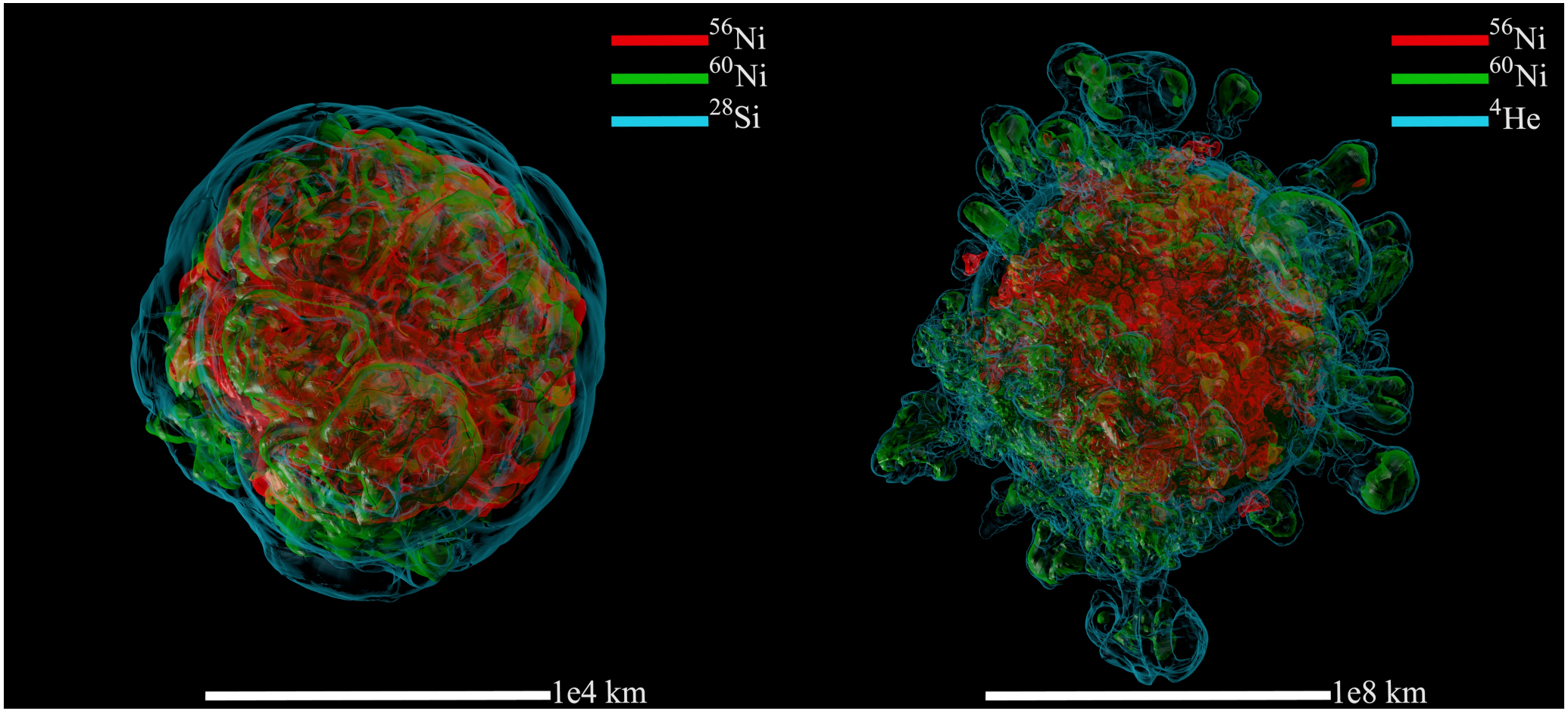
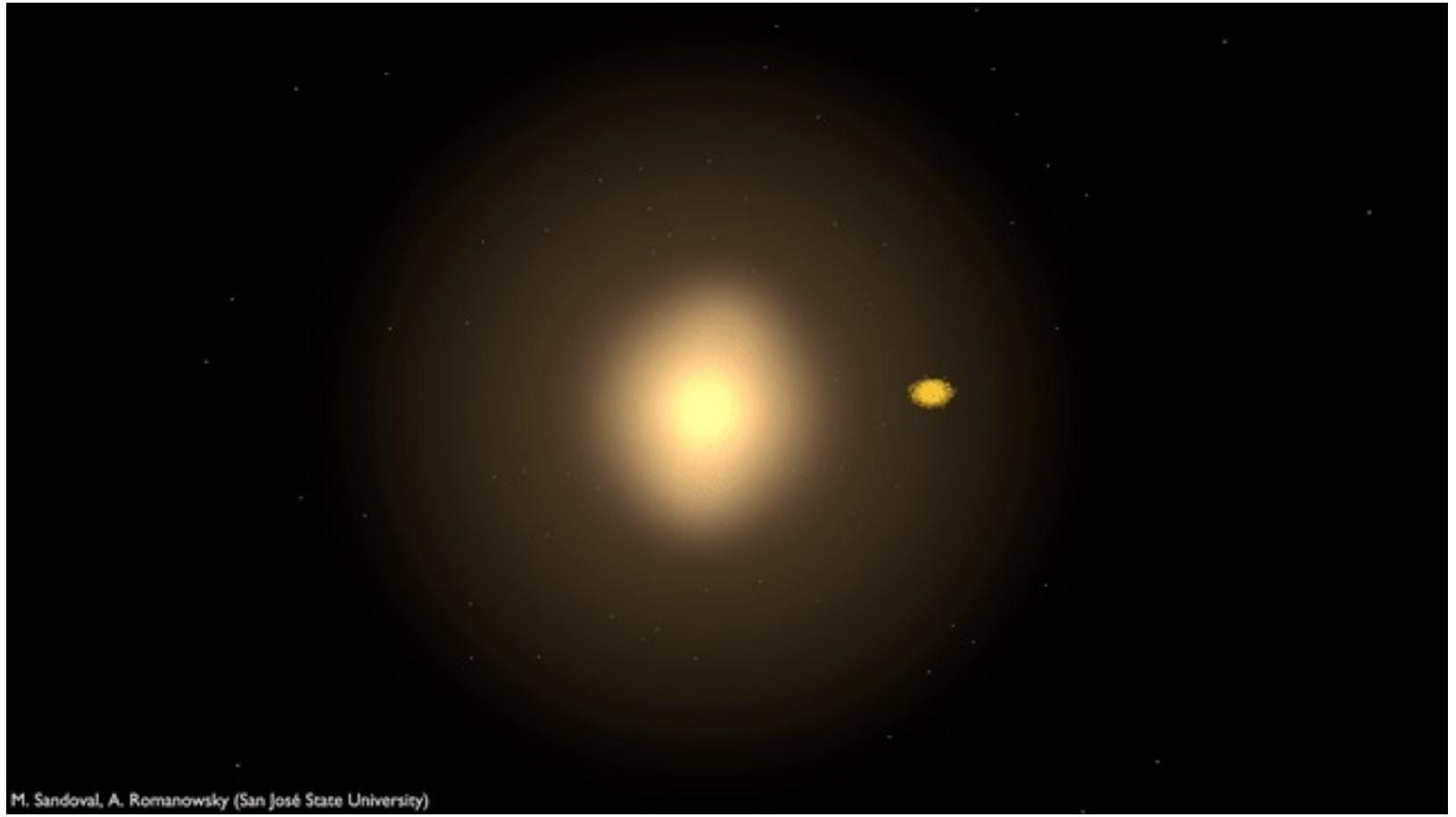


Image credit: Michael Sandoval

Simulations



You will get to make a version of this next week!

Image credit: Michael Sandoval

Now…for the Python…

The Basics – Part One

Interactive Mode

- Good place to start
- First:

  $ cd ~/foundational_hpc_skills/intro_to_python
- Then:

  $ python3

```
$ python3

Python 3.7.6 (default, Aug 13 2021, 16:40:31)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

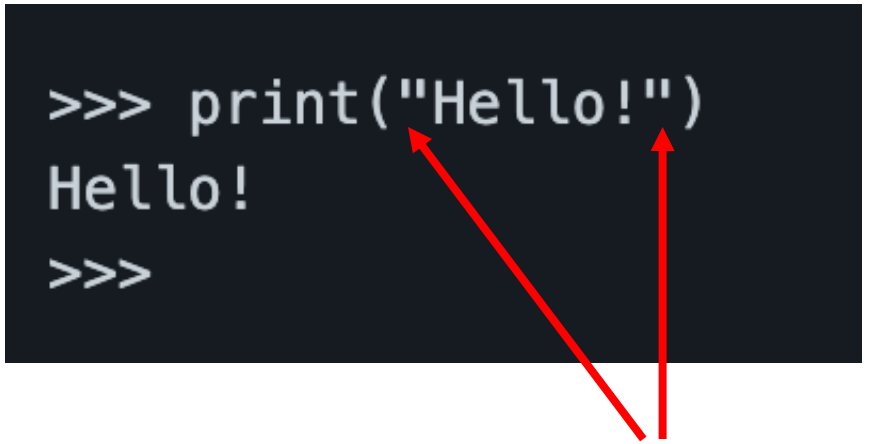
Line Syntax

- The **>>>** symbol is the prompt of the interactive Python interpreter
  - Anything you type in, will be executed with Python
- Executed on a line-by-line basis → a line will be executed after pressing Enter/Return.
  - You do not have to type in a semicolon, you just need to press Enter/Return.
- If you need to quit for whatever reason, type: quit()

```
$ python3

Python 3.7.6 (default, Aug 13 2021, 16:40:31)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Line Syntax: Using the print Function I

- First, let's test things out by making Python say "Hello!" to us using the **print** function:
  - Call functions by using parentheses: **FUNCTION_NAME(function arguments)**



- Enclose what you want to print in a pair of “
- We will talk more about the “’s later…

Line Syntax: Using the print Function II

- Now displaying **>>>** again and waiting for you to tell it what to do next
- Let's use the print function again to print out multiple things at once that are side-by-side
- Print multiple things by using a comma between statements

```
>>> print("I am on the left!", "I am on the right!")
I am on the left! I am on the right!
>>>
```

Line Syntax: Breaking and Continuing Lines

- Lines may begin with **...** (called "continuation lines")
  - Indicates that Python is waiting for you to input something else or "finish" the line
- To see this, we can explicitly span a single Python statement across multiple lines by using the **** symbol at the end of a line.

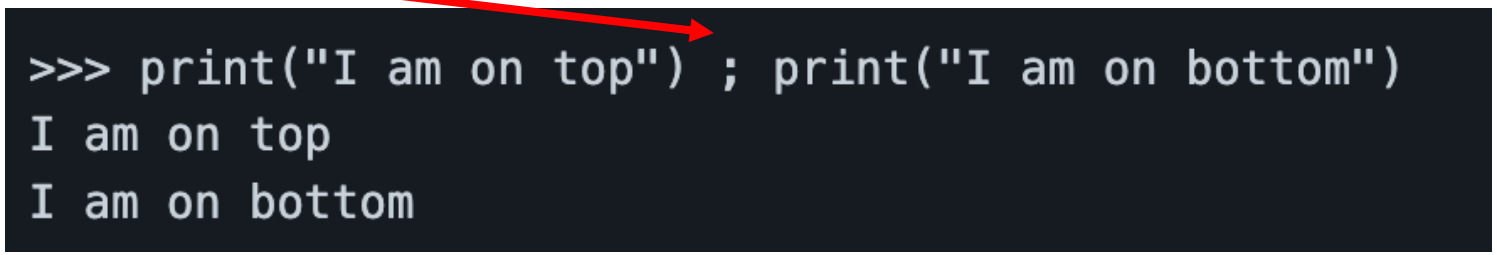
```
>>> print("I am on the left!", \
... "I am on the right!")

I am on the left! I am on the right!
```

Breaks a line

Beginning of a continuation line

Line Syntax: Combining Lines

- Now let's try to combine multiple statements into a single line.
- We can combine both statements into a single line using the ; symbol:



```
>>> print("I am on top") ; print("I am on bottom")
I am on top
I am on bottom
```

- To the Python interpreter, the above example would just look like:

```
>>> print("I am on top")
>>> print("I am on bottom")
```

Line Syntax: Comments

- Comments are not interpreted by Python and are used to clarify code
  - usually acting as notes to yourself or others to explain code
  - you do NOT need to type out comments you see in this guide
- Use the **#** symbol:

```
>>> # this is a comment
>>> print("Comments are hidden") # this is also a comment
Comments are hidden
>>> print(" # this is not a comment because enclosed in quotes")
 # this is not a comment because enclosed in quotes
```

Numbers and Variables

- Now that we have covered some of the basic syntax, let's dig deeper and show off what Python can do with numbers…
- Python commonly used as a calculator
  - use the operators **+,-,*,/** for addition, subtraction, multiplication, and division

```
>>> 2 + 2         # addition
4
>>> 0 - 1         # subtraction
-1
>>> 2 * 3         # multiplication
6
>>> 10 / 5        # division
2.0
>>> 3 ** 2        # exponents/powers (e.g., 3 to the power of 2)
9
>>> (1 + 2) * 4   # parentheses have priority
12
```

Numbers and Variables: Integers and Floats

- integers (-2, -1, 0, 1, etc.) are of type "**int**"
- rational numbers (-2.0, -1.3, 0.2, 1.4, etc.) are of type "**float**"

```
>>> type(1.0) # check type of 1.0
<class 'float'>
>>> type(1)   # check type of 1
<class 'int'>
>>> 4.2 * 2   # mixed inputs results in a float answer
8.4
>>> 17 / 3    # integer division results in a float answer
5.666666666666667
>>> 17 // 3   # integer division with "//" floors the answer to an integer
5
```

Numbers and Variables: Assigning Variables I

- A variable name can only contain alpha-numeric characters and underscores (A-Z, 0-9, and _) and cannot start with a number.
- You can use the equal sign **=** to assign a value to a variable:

```
>>> width = 2.0  # storing a value into a variable called "width"
>>> height = 3.0 # storing a value into a variable called "height"
>>> area = width * height
>>> print(area)  # explicitly print out a variable
6.0
>>> area         # prints out a variable (only possible in interactive mode)
6.0
```

Numbers and Variables: Assigning Variables II

- A variable name can only contain alpha-numeric characters and underscores (A-Z, 0-9, and _) and cannot start with a number.
- Variables are case-sensitive and can't have spaces!

```
>>> print(area)      # print out our old variable
6.0

>>> print(Area)      # slightly misspell our variable (case-sensitive)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Area' is not defined

>>> width new = 4.0  # try to set a variable name with a space in it
  File "<stdin>", line 1
    width new = 4.0
              ^
SyntaxError: invalid syntax
```

Numbers and Variables: Assigning Variables III

- For a more advanced approach, here are some handy tricks when using variables:

```
>>> x, y, z = 1, 2, 3  # set multiple variables at once to different values
>>> print(x,y,z)
1 2 3
>>> a = b = c = 4      # set multiple variables at once to the same value
>>> print(a,b,c)
4 4 4
>>> e = 5
>>> f = 6
>>> print(e,f)
5 6
>>> e, f = f, e        # swap two variables
>>> print(e,f)
6 5
```

Numbers and Variables: Assigning Variables IV

- Magic _ variable:

```
>>> tax = .10       # set tax to ten percent
>>> price = 100.0   # price is $100
>>> tax * price     # additional price, gets assigned to "_" variable
10.0
>>> _               # print out what "_" currently is
10.0
>>> price + _       # total price, also resets "_" variable
110.0
>>> _
110.0
```

Strings and Slicing

- A "**string**" in Python is a sequence of characters that usually represent a form of text
- In other coding languages, such as Fortran and C, text is usually stored in some form of "character" or "char" datatype
  - Python does not have a character datatype
- A single character in Python is simply a "string" datatype called str with length 1 (a string with two characters would be of length 2, etc.).
- Like with numbers, Python also can manipulate strings…

Strings and Slicing: Strings as Variables

- You can assign strings to variables using **=**
- To create a string in Python, use a pair of single quotes '...' or double quotes "..."
  - Personal preference, but can be useful in certain scenarios

```
>>> x = "Hello"                          # double quote example
>>> y = 'Hello'                          # single quote example
>>> x==y                                 # check if "x" is equivalent to "y"
True
>>> no_error_1 = "You're cool"           # double quotes enclosing a single quote
>>> error_1 = 'You're cool'              # using single quotes instead (error)

  File "<stdin>", line 1
    error_1 = 'You're cool'
                     ^
SyntaxError: invalid syntax

>>> no_error_2 = ' No "i" in "Team" ' # single quotes enclosing double quotes
>>> error_2 = " No "i" in "Team" "    # using double quotes instead (error)

  File "<stdin>", line 1
    error_2 = " No "i" in "Team" "
                     ^
SyntaxError: invalid syntax
```

Strings and Slicing: Manipulating Strings

- Similar to numbers, strings also can be manipulated using the **+** and ***** operators
- The **+** operator combines two strings
- The ***** operator can be used to repeat a string

```
>>> "gg" + "ez"                # combining two strings using +
'ggez'
>>> 3 * "yes"                  # repeat a string using *
'yesyesyes'
>>> ( 3 * "yes" ) + "no"       # repeat a string and combine
'yesyesyesno'
```

Strings and Slicing: Indexing Strings I

- As you will see with “lists” later, one cool thing about strings is that you can "slice" them to extract specific portions of the string
- Slicing is heavily used in Python programming, especially when analyzing data, plotting, and sorting
- "Indexing" is a subset of slicing that only extracts one character from a string, while "slicing" can extract multiple characters at once (i.e., multiple indices)

Strings and Slicing: Indexing Strings II

- Python counting starts at 0 instead of 1
  - i.e. counting goes like: 0,1,2,3,4 **NOT** 1,2,3,4,5
  - Can be confusing at times
- Slicing strings works the same way
  - The first character in a string has index 0, the second character has index 1, etc.
- Let's see this in some examples…

Strings and Slicing: Indexing Strings III

- To start, you must put indices in square brackets **[ ]**:

```
>>> z = "abcdefghij"  # make a test string of length 10
>>> len(z)
10
>>> z[0] # character in index/position 0
'a'
>>> z[1] # character in index/position 1
'b'
>>> z[9] # character in index/position 9
'j'
```

the “len” function finds full length of string

- In our example we have 10 letters, but since indexing starts from 0, the final letter in the string is index 9.

Strings and Slicing: Indexing Strings IV

- You can also use negative indices to start at the end of the string
  - i.e. negative indices reads from "right to left" instead of "left to right"

```
z = "abcdefghij"
```

```
>>> z[-1]  # character in position -1 (equivalent to position 9)
'j'
>>> z[-2]  # character in position -2 (equivalent to position 8)
'i'
```

- Negative indices can be helpful when you don't know how long a string is, but when you KNOW you want the end of it

Strings and Slicing: Slicing Strings I

- You can use the **:** operator inside of **[ ]** to pick a range of indices (inserted on either side of the **:**).
  - Note that when using : the left side is *inclusive* while the right side is *exclusive*
  - i.e. the syntax is **[desired start index : desired end index + 1]**

```
z = "abcdefghij"
```

```
>>> z[0:10]       # slices indices 0-9 (the entire string of length 10)
'abcdefghij'
>>> z[0:5]        # slices indices 0-4 (the first five characters)
'abcde'
>>> z[1:5]        # slices indices 1-4
'bcde'
>>> z[1 : (4+1)]  # slices indices 1-4 (same as above, but with math)
'bcde'
```

Strings and Slicing: Slicing Strings II

- You can choose to leave a side of the **:** blank
- Leaving the left side blank: slice begins at the start of the string
- Leaving the right side blank: slice stops at the end of the string
- Leaving **both** sides blank (i.e., **[:]**): slices ENTIRE string

```
>>> z[:]    # return the entire string
'abcdefghij'
>>> z[2:]   # slice from index 2 to the end of the string
'cdefghij'
>>> z[:8]   # slice from the beginning of the string until index 7
'abcdefgh'
>>> z[:-1]  # slice the entire string ignoring the last character
'abcdefghi'
>>> z[:-2]  # slice the entire string ignoring the last two characters
'abcdefgh'
```

Strings and Slicing: Slicing Strings III

- Because of the left side of **:** being included and the right side being excluded, this makes sure that **z[: index] + z[index :]** is always equal to z

- The techniques we use to manipulate strings are also used for other things, which is what we'll be covering next.

```
z = "abcdefghij"
```

```
>>> z[:2]
'ab'
>>> z[2:]
'cdefghij'
>>> z[:2] + z[2:]
'abcdefghij'
>>> z[:4] + z[4:]
'abcdefghij'
```

Lists

- Now that you know a few of the micro-scale datatypes, the next thing we are going to cover is how to group data
- Lists are defined as a "compound" datatype and are able to group together both numbers and strings, either individually or in a mix.
- For people already familiar with arrays…pretty similar

Lists: Lists as Variables

- The data contained in a list are comma separated and enclosed by square brackets
- Here are some examples:

```
>>> blank_list = []                        # create a "blank" list of length 0
>>> tiny_list = ['hola']                   # create a list of length 1
>>> num_list = [0, 1, 2, 3.0, 4.0]         # create a list of numbers of length 5
>>> str_list = ['hey', 'hi', 'yo']         # create a list of strings of length 3
>>> mixed_list = [1, 7, 'cool', 'groovy'] # create a mixed list of length 4
```

- We will be using the above lists in the next couple examples, so be sure not to overwrite them!

Lists: Using Lists

```
>>> blank_list = []
>>> tiny_list = ['hola']
>>> num_list = [0, 1, 2, 3.0, 4.0]
>>> str_list = ['hey', 'hi', 'yo']
>>> mixed_list = [1, 7, 'cool', 'groovy']
```

- Lists obey some of the same operations we covered for strings, such as combining and slicing:

```
>>> combined_list = num_list + str_list  # combining two lists into a new list
>>> combined_list[:]                     # return/slice entire list
[0, 1, 2, 3.0, 4.0, 'hey', 'hi', 'yo']
>>> combined_list[0]                     # return/slice first entry in list
0
>>> combined_list[-1]                    # return/slice final entry in list
'yo'
>>> combined_list[0:5]                   # return/slice first 5 entries
[0, 1, 2, 3.0, 4.0]
>>> len(combined_list)                   # return length of the list
8
```

Lists: Slicing Lists I

- Unlike with strings, you can modify a specific entry in a list with slicing/indexing:

```
>>> squares = [0, 1, 4, 9, 16, 25, 'temp'] # create a list of "squared" values
>>> squares[6]                              # final entry does not fit pattern
'temp'
>>> squares[6] = 36                         # modify data of the final entry
>>> squares                                 # inspect the modified list
[0, 1, 4, 9, 16, 25, 36]
```

Lists: Slicing Lists II

- This slicing ability can also be used to modify sections of lists, or be used to completely remove sections:

```
>>> dummy = ['a', 'b', 'c', 1, 2, 3, 'g', 'h', 'i'] # create a new list
>>> dummy[3:6]                                      # identify problem area
[1, 2, 3]
>>> dummy[3:6] = ['D', 'E', 'F']                    # replace problem area
>>> dummy                                           # inspect modified list
['a', 'b', 'c', 'D', 'E', 'F', 'g', 'h', 'i']
>>> dummy[3:6] = []                                 # decide to remove that area
>>> dummy                                           # inspect the list again
['a', 'b', 'c', 'g', 'h', 'i']
>>> dummy[:] = []                                   # remove all entries
>>> dummy
[]
```

End of Part One

- This concludes the first half of the basics -- the "interactive shell" portion
- Although what's covered next is also considered part of "the basics," the topics are better suited in a scripting environment.
- We will be transitioning out of the Python interactive shell into writing and running Python scripts instead.
- To exit interactive mode:

```
>>> quit()
```

The Basics – Part Two

Scripts

- Pre-made scripts located in intro_to_python/ directories
- All that we covered previously still applies when running Python scripts, but there are a few changes
  - No “>>>” at beginning of lines
  - Similarly, no “...” at the beginning of continuation lines
  - Gain the ability to comment out multiple lines at once using '''', which is a nice bonus
  - Everything else discussed goes unchanged, yay!

Scripts: Running Scripts

- To run Python scripts using Python 3, it's as simple as executing “python3 file.py” on the command line
- Although most of the scripts are pre-made, it is a good exercise to make your own first
  - A blank file called first_script.py is included for you to fill out
  - Use VIM

```
$ python3 first_script.py
```

```
Hello from your first script!
Jenny has 11.765 apples
```

```
# top of the file

'''
Showing off commenting out multiple lines.
Sometimes comments of this nature are used
to provide a description of the file at
the top of the file.
They can span multiple lines when enclosed
by a pair of three single quotes.
'''

print("Hello from your first script!")

# add some numbers
x = 8.675
y = 3.09
z = x + y

# print out some output
print("Jenny has ", z, " apples")
```

Loops and Indentation: "For" Loops

- Relevant directory: ~/foundational_hpc_skills/intro_to_python/loops
- Loops make Python iterate over a certain section of code (usually over a list or a set of numbers)
  - one of these is called a "for" loop
- A for loop iterates over the items of a sequence in the order that the items appear in the sequence
- The body of a "for" loop is separated from the rest of the code using indentation
  - Indentation is a way of grouping statements, which is done by using either tabs or spaces (personal preference, but be consistent)

```
for value in sequence:
    body of the loop
```

Loops and Indentation: "For" Loop Example 1

- In actual English, you can think of the line "**for i in loop_list:**" as meaning: "**For every entry (i) in loop_list, Python will do the following:**"

```
# for_loop_list.py

# create a list to loop over
loop_list = ['cat', 'dog', 'hamster', 'bird']

# use a "for" loop to loop over our list
# print out each entry "i" in "loop_list"
for i in loop_list:
    print(i)
```

```
$ python3 for_loop_list.py
cat
dog
hamster
bird
```

Loops and Indentation: "For" Loop Example 2

- The for-loop can also be used to iterate over a range of values
  - helpful for indexing and slicing in a loop
- To iterate over a range of values, you can use the range function, which takes a start value, a stop value, and an optional step value: **range(start, stop, step)**
  - Just like with indexing, the start value is inclusive while the stop value is exclusive.

Loops and Indentation: "For" Loop Example 2 II

```
# for_loop_range.py


# iterate from 0 through 10 (exclude 10) in steps of 1
print( 'Example 6.1: range(0,10,1)' )

for i in range(0,10,1):
    print(i)


# iterate from 0 through 10 (exclude 10) in steps of 2
print(' ')
print( 'Example 6.2: range(0,10,2)' )

for i in range(0,10,2):
    print(i)


# if only supplying one number, it will iterate up to that number
print(' ')
print( 'Example 6.3: range(4)' )

for i in range(4):
    print(i)


# iterate over a list using its length and indexing
print(' ')
print( 'Example 6.4: range(length_x)' )

x = ['O','L','C','F']
length_x = len(x)
for i in range(length_x):
    print(x[i])
```

```
$ python3 for_loop_range.py
Example 6.1: range(0,10,1)
0
1
2
3
4
5
6
7
8
9

Example 6.2: range(0,10,2)
0
2
4
6
8

Example 6.3: range(4)
0
1
2
3

Example 6.4: range(length_x)
O
L
C
F
```

Loops and Indentation: “While” Loops

- The second type of loop is called a "while" loop, which keeps iterating until a certain condition is no longer true
- The syntax is a bit simpler with the "while" loop:

```
while condition:
    body of the loop
```

- You can think of the line “**while condition:**” as meaning: "**While this condition is true, Python will execute the following:**"

Loops and Indentation: “While” Loops II

- Because the "while" loop constantly checks to see if a certain statement is "true", the loop is typically used with logical **comparison operators**
- The standard comparison operators are:
  - < (less than)
  - > (greater than)
  - == (equal to)
  - <= (less than or equal to)
  - >= (greater than or equal to)
  - != (not equal to)

Example:

1 < 3 returns True
1 > 3 returns False

Loops and Indentation: "While" Loop Example

```
# while_loops.py


# iterate only while "a" is less than "b"
print('Example 6.5: while a<b')

a = 0
b = 5
while a<b:
    print('a equals', a)
    a = a + 1

# iterate until "c" equals 6
print(' ')
print('Example 6.6: while c!=6')

c = 0
while c!=6:
    print("Am I 6 yet?", "Nope, I am", c)
    c = c + 1
```

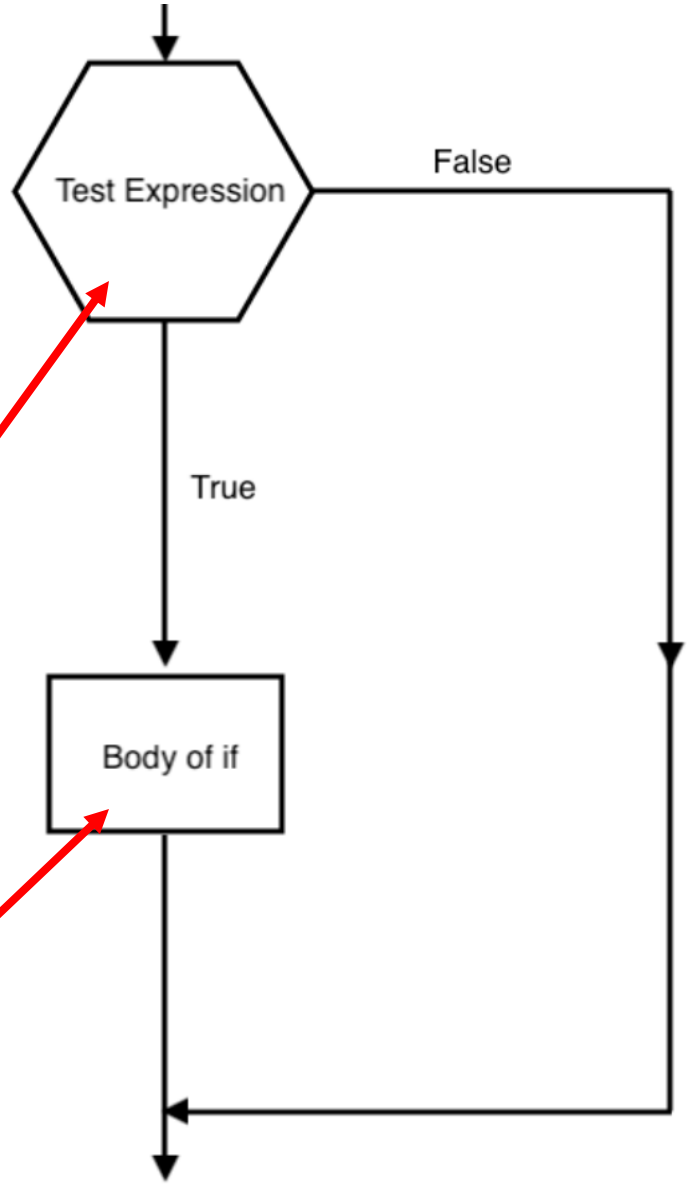
```
$ python3 while_loops.py
Example 6.5: while a<b
a equals 0
a equals 1
a equals 2
a equals 3
a equals 4

Example 6.6: while c!=6
Am I 6 yet? Nope, I am 0
Am I 6 yet? Nope, I am 1
Am I 6 yet? Nope, I am 2
Am I 6 yet? Nope, I am 3
Am I 6 yet? Nope, I am 4
Am I 6 yet? Nope, I am 5
```

Loops and Indentation: “While” Loop Warning

- "While" loops can be dangerous at times, as it is very easy to accidentally get stuck in an infinite loop
- In the previous examples, if we didn’t modify either “a” or “c”, their value never would have changed and we would have gotten stuck in the loop
- In general, if you can convert a "while" loop into a "for" loop, it is recommended to use the "for" loop version instead
- Now that we've introduced conditions being either "True" or "False" in the context of loops, the next step is to introduce how to section off your code based on "True" or "False" conditions

if-elif-else Statements: The "if" Statement

- Relevant directory:
  ~/foundational_hpc_skills/intro_to_python/if_elif_else
- If you want to execute a piece of code only if a specific condition is met, the "**if**" statement comes in handy
- For example, maybe you only want to print something out if a specific variable is "True", or when a math expression yields a specific value




```
if condition:
    body of if statement
```

if-elif-else Statements: if Statement Example

- Recall from earlier that the **==** operator checks if two things are equal
- x did indeed equal 1, so the condition was "True" and Python executed the print function
- x did **NOT** equal 2, so the condition was "False" and did not execute the if statement
  - Because we did not tell Python what to do if the condition was "False", Python did nothing and skipped over it.

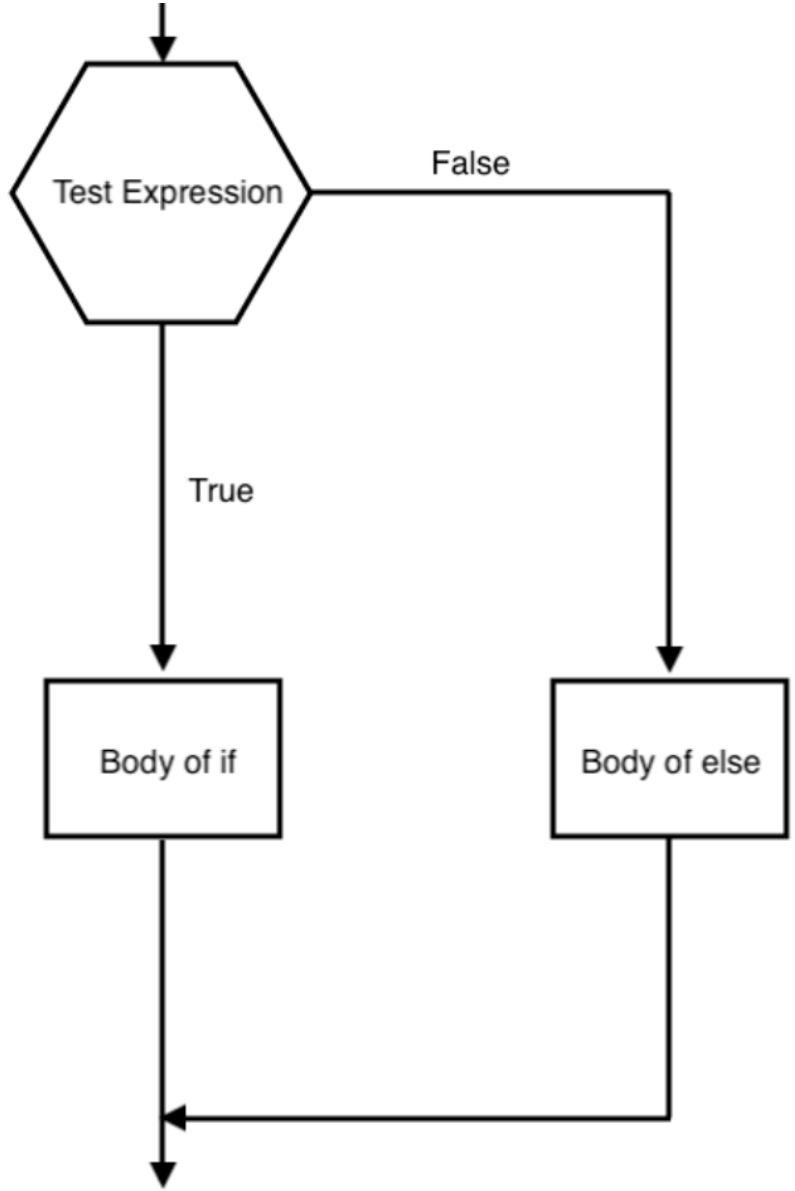
```
# if.py

# initialize a variable "x"
x = 1

# Example 7.1: check to see if "x" is equal to 1
if x==1 :
    print('x is 1')

# Example 7.2: check to see if "x" is equal to 2
if x==2 :
    print('x is 2')
```

```
$ python3 if.py
x is 1
```

if-elif-else Statements: The “else” Statement

- To explicitly tell Python what to do if the condition is "False", you need to add an else statement

```
if condition:
    body of if statement     # Do this if the condition is True
else:
    body of else statement   # Do this if the condition is False
```



if-elif-else Statements: else Statement Example

- Python only executes one of the print statements
- The x variable was equal to 2, so the "**if x==1**" condition was False and Python executed the **else** statement instead

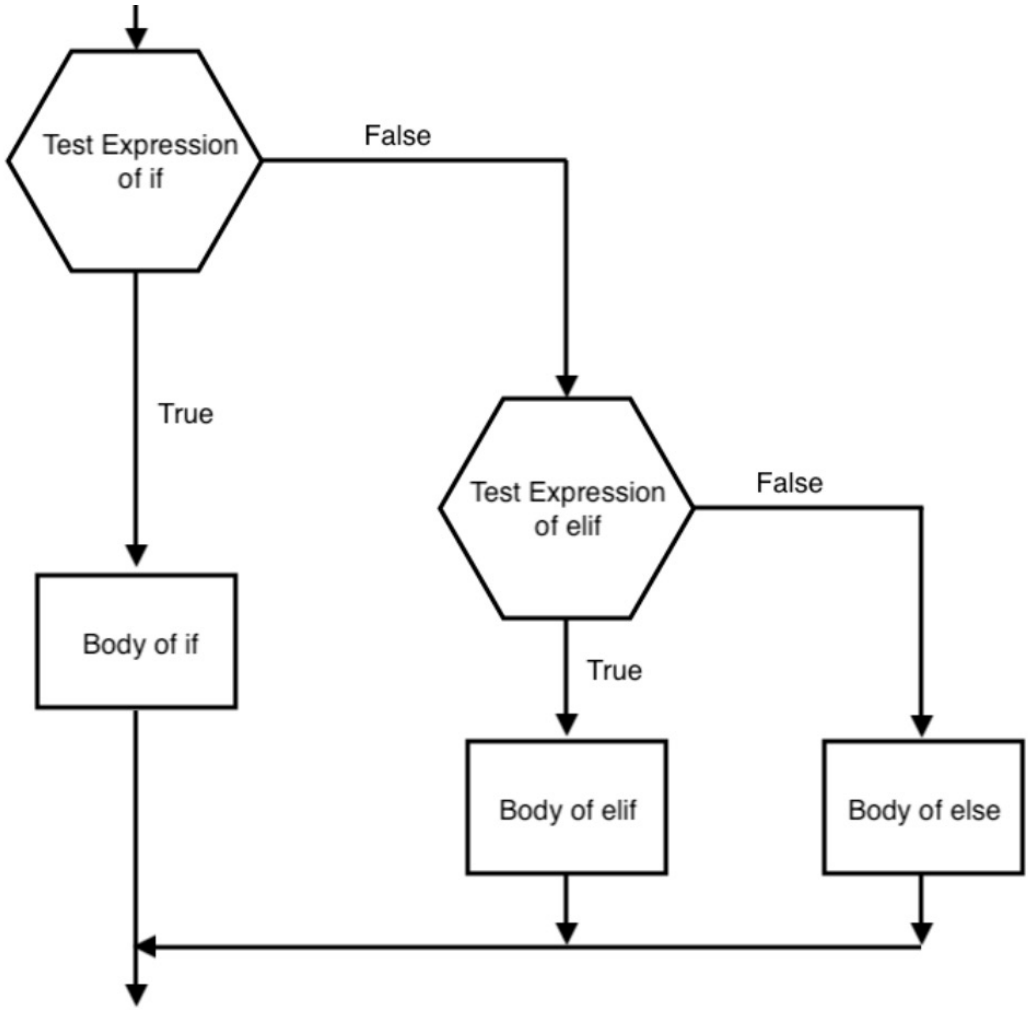
```
# else.py

# initialize a variable "x"
x = 2

# check to see if "x" is equal to 1, else do something different
if x==1 :
    print('x is 1')
else:
    print('x is not 1')
```

```
$ python3 else.py
x is not 1
```

if-elif-else Statements: The “elif” Statement

- Perhaps you want to tell Python to check for multiple conditions, instead of just one
  - You need to add an “**elif**” statement (short for "else-if")

```
if condition_one:
    body of if statement    # Do this if condition_one is True
elif condition_two:
    body of elif statement  # Do this if condition_two is True
else:
    body of else statement  # Do this if both conditions are False
```



if-elif-else Statements: elif Statement Example

- Because Python skips the remaining if-elif-else code once it finds a "True" condition, only **ONE** of the statements will be executed across an entire if-elif-else section of code
  - Even if multiple statements are "True", it will only execute whichever one it finds first, starting from the top

```
# elif.py

# initialize a variable "x"
x = 1

# check to see if "x" is less than 1, 2, or 3
if x<1 :
    print('x is less than 1')
elif x<2 :
    print('x is less than 2')
elif x<3 :
    print('x is less than 3')
else:
    print('x is not less than 1, 2, or 3')
```

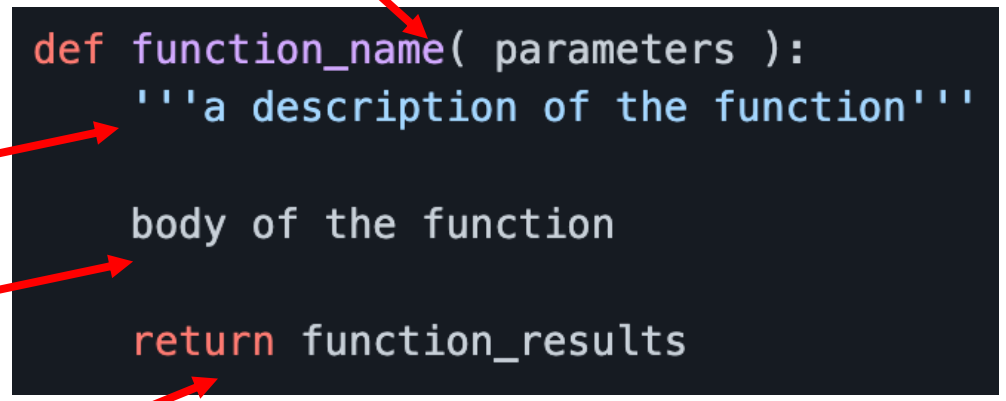
```
$ python3 elif.py
x is less than 2
```

Functions

- Relevant directory: ~/foundational_hpc_skills/intro_to_python/functions
- A function is a block of code that only runs when it is called, but is designed to be reusable
  - Helpful for trying to repeat a specific action within your code
- We already used some pre-defined functions (**print** and **len**), but you can also create your *own* functions

Functions: Structure of a Function

- The function block begins with **def** keyword followed by the desired name of the function with a set of parentheses ( )
- Any input parameters for the function are placed inside the parentheses next to the function name
- The indented section:
  - A description of the function enclosed by a set of triple-quotes (optional but good habit)
  - The "body" of the function is the code you want the function to perform when called
  - The return statement exits the function, and can pass back a specific variable or expression, usually the results of the body



Functions: Function Example

- Function returns a variable “z”
  - Won’t automatically print on its own (Ex. 8.1)
  - Must be assigned to a variable and then printed (Ex. 8.2)
- sum_var(x=3, y=5) syntax also works

```
# func_var.py

def sum_var(x,y):
    '''This function will sum
       x and y into a variable
       z and return the variable z
       when the function exits'''

    z = x + y

    return z

# Example 8.1: execute the function without saving the result
sum_var(3,5)

# Example 8.2: execute the function while saving result to a variable
test = sum_var(3,5)
print(test)
```

```
$ python3 func_var.py
8
```

Closing Remarks

- A long and bumpy ride, but we made it!
- This isn't all that python has to offer, just here to give you some building blocks / foundation
  - Notable thing (sort of) that I skipped: **classes**
    - Not the most important thing as a building block, but may encounter later
- Everything here was condensed, so I encourage you to go out and explore some more yourself online
  - User Input
  - Variables in strings / formatting strings

Additional Resources

- Python Website: https://www.python.org/
- Python Documentation https://docs.python.org/3/
- Python Tutorial: https://docs.python.org/3/tutorial/introduction.html
- Including Variables in Strings: https://realpython.com/python-f-strings/
- User Input Tutorial https://www.geeksforgeeks.org/python-3-input-function/

Homework

- A list of challenges with descriptions/instructions are available in the intro_to_python README (https://github.com/olcf/foundational_hpc_skills/tree/master/intro_to_python)
  - Challenges vary in difficulty
- To access the challenges on the command line, you must change directories to the challenges directory:

  $ cd ~/foundational_hpc_skills/intro_to_python/challenges
- If you are having trouble completing the challenges, potential solutions are provided in the solutions sub-directory