ITEC 1150 Week 6 Chapter 3

FUNCTIONS, SCOPE, EXCEPTION HANDLING

Functions

A review of functions we've used, and how to create our own

- A **Function** is a reusable piece of code.
- A Function
  - Can (optionally) take parameters
  - Can (optionally) return values
  - Can be packaged so it can be **imported** into other programs
  - Can be packed into libraries (called modules in Python) which can be downloaded so the functions can be imported

Functions we've already seen

- `print` takes zero to many parameters, as we saw in Chapter 1. It does not return a value (technically, it returns `None`):

```
print("Hello world")
```

- `input` takes zero or one parameter (the prompt to the user) and returns a string:

```
input("What is your age?")
```

- `int` takes one parameter (the string to be converted) and returns a number. If it cannot turn the string into a number, it **throws** an **exception**.

```
int("4")
```

- These functions (and many more) are built-in to Python.

Why Programming Languages Have Functions

Suppose that as part of a program you needed to make sure a String was all uppercase because user input was displayed as headings on a table and needed to be uppercase.

The code to do so is on the right.

If you read the code, you'll see that not only is it long, but it includes things you may not understand.

Having to learn all about ASCII and how to convert characters probably isn't the main problem you're trying to solve--converting to uppercase for display is one small detail in your program.

```
lowercase = input('Enter a string of lowercase letters: ')
uppercase = '' # initialize a new empty string variable, to build up a new
string
for char in lowercase:
    ascii = ord(char)
    # test if char is a lowercase letter & if so, reassign to upper
    if ascii >= 97 and ascii <= 122:
        ascii = ascii - 32
        new_char = chr(ascii)
        uppercase = uppercase + new_char
    else:
        uppercase = uppercase + char
print('The string in uppercase is ' + uppercase)
```

What if you had to include this much code each time you need to turn a lowercase string into uppercase?

Pre-Made Functions Are Very Useful

```
lowercase2 = input('Enter some lowercase letters: ')
uppercase2 = lowercase2.upper()
print(uppercase2)
```

Calling this function is *a lot* easier than writing all the code on the previous slide!

- `upper()`, `lower()`, `print()` and other built-in functions and methods are easy, safe, reliable and readable!
- Someone already wrote, tested, and debugged them.
- There are hundreds more!

Methods – Functions Attached to Objects

- The functions we've used so far are standalone functions, but there are others which are attached to objects themselves. For example, `.upper()` is a function that is attached to a String—you must call `'abc'.upper()`, not `upper('abc')`:

```
lowercase2 = input('Enter some lowercase letters: ')
uppercase2 = lowercase2.upper()
print(uppercase2)
```

- `.upper()` is a **method**, which is what we call a function that is part of a **Class** (in this case, `.upper()` is a method of the String class).
- We won't get into Classes in this course, but a short explanation is that a Class is a way to combine data *and* functions into a reusable unit of code.

Writing Functions

WE'RE NOT LIMITED TO THE ONES THAT COME WITH PYTHON

WE CAN MAKE OUR OWN!

Advantages of Writing Your Own Functions

- As you have seen, functions and methods can gather long code blocks to perform specific tasks like capitalizing every character in a string.
- Functions help organize your program and make it more understandable.
- They eliminate repetitive code: if you define a function once, you can call it many times.
- Debugging is easier – you can test and improve each function individually.
- Once you've defined a function, you can even call it in other programs.

The None Data Type

- Before jumping into writing functions, we'll introduce a new data type that we mentioned earlier.
- `None` is a special data type that is used to indicate there is no value.
- That might seem weird, but it is sometimes necessary to check if your variable has data in it, or to create a variable that will have data later but doesn't yet.
- It is often used when parameters for a function are optional, as seen below (we'll look at that syntax in a bit).

```python
def print_greeting(name=None):
    if name is None:
        print('Hello world!')
    else:
        print(f'Hello, {name}!')
```

Defining a Function

- A function always starts with `def`
- Then the function name (`calc_area` in this case)
- Then parenthesis, which contain any parameters the function needs (`length` and `width` in this case)
- Then the body, which is indented in a block
- A function always returns a value
  - You can specify what to return
  - If you don't specify a value, the function returns `None`

```
def calc_area(length, width):
    area = length * width
    return area
```

Using Our Simple Function

```
def calc_area(length, width):
    area = length * width
    return area

width = int(input("What is the width in whole numbers? "))
length = int(input("What is the length in whole numbers? "))
rect_area = calc_area(length, width)
print(f'The area of a {length} by {width} rectangle is {rect_area}') # OR
print(f'The area of a {length} by {width} rectangle is {calc_area(length, width)}')
```

- We can use the value returned by a function by putting it into a variable, or even putting it right into a format string.

Returning Values From Functions

- To return a value from a function, use the return keyword followed by the value you want to return (line 3).
- Note that you can return the result of an expression; it doesn't have to be a variable:

```
return length * width
```

- The return value of a function can be assigned to a variable—on line 6, the print statement outputs the value returned by the function.

```
1. def rectangle_area(length, width):
2.     area = length * width
3.     return area
4.
5. rectangle1 = rectangle_area(2, 3)
6. print(rectangle1)
```

Returning Multiple Values

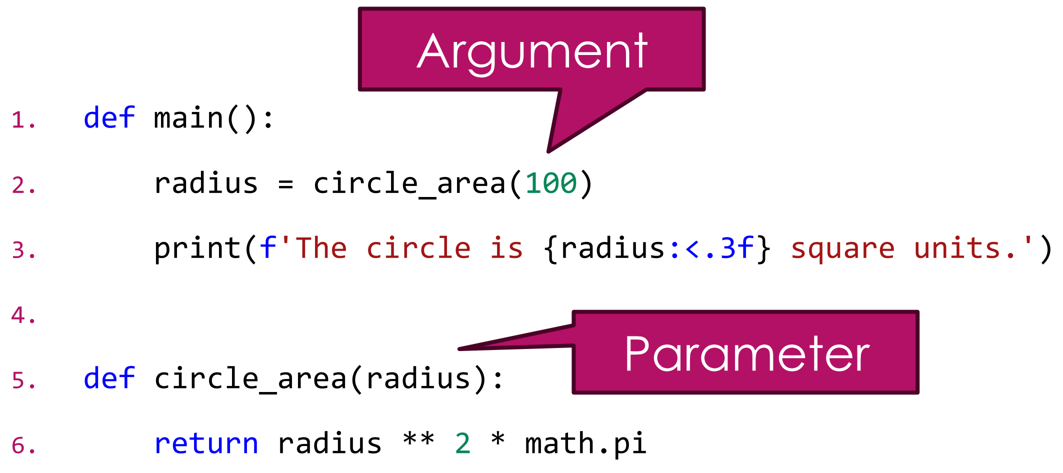
```python
import math

def main():
    area, volume = calculate_cylinder(3, 4)
    print(f'The area of the face of the cylinder is {area:.2f} square units.')
    print(f'The volume of the cylinder is {volume:.2f} cubic units.')

def calculate_cylinder(radius, height):
    face_area = math.pi * radius**2
    volume = face_area * height
    return face_area, volume

main() # function call
```

Putting two returned values into variables. This is done by order, not by name!

Returning two values

Parameters and Arguments

- Data the function needs to receive = **parameters**
- Data you send to the function = **arguments**
- It's not critical to remember the difference between parameters and arguments. This explanation is just so you're not confused if you hear these terms.



```python
def main():
    radius = circle_area(100)
    print(f'The circle is {radius:<.3f} square units.')

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

Multiple Parameters In a Function

```python
def main():
    string_data = input('Please enter a string: ')
    repeat = int(input('How many times do you want to repeat the string? '))
    string_repeater(string_data, repeat) # function call

def string_repeater(text, n):
    repeated_string = (text + ' ') * n
    print('Here\'s your repeated string...')
    print(repeated_string)

main() # function call
```

- When you call a function, match your arguments to the function definition in 3 ways:
  - the <u>number</u> of parameters
  - the <u>order</u> of parameters
  - the <u>type</u> of parameters.

Documenting Functions

- Give them a meaningful name that describes what the function does
- Write comments which explain the purpose and operation of the function.
- The docstring in the function provides important details that make it easy for someone to use the function without having to read the code!
- Documenting the return value(s) is good practice.

```
def get_pos_int():
    """
    Gets a positive integer from a user. Prompts the user for an
integer, validates that it's a positive integer, and will not
return until a valid entry is received.
    @return: a positive integer
    """
    pos_int = input('Please enter a whole number: ')
    while pos_int.isnumeric() is False or int(pos_int) <= 0:
        pos_int = input('Enter a whole number greater than 0: ')
    pos_int = int(pos_int)
    return pos_int
```

Calling Functions

- A function won't run unless it is *called*.
- The `rectangle_area` function is called on line 5; if this line wasn't present, none of the code in the function would run.

```
1. def rectangle_area(length, width):
2.     area = length * width
3.     return area
4.
5. rectangle1 = rectangle_area(2, 3)
6. print(rectangle1)
```

Function Ordering in Code

- The only strict rule is that functions must be declared before they are called by any code outside a function. The first listing on the right will not run, because `main()` is called *before* the `main()` function is declared.

- The second listing is fine, because the call to `main()` happens after `main()` has been declared.
- Note that the call within `main()` to `print_welcome()` (on line 2) is also OK, because that will not run until `main()` is called!
- In other words, if the call to `main()` is at the very end of the program, and the main function sets off a valid chain reaction of function calls and other commands, the order of the other function definitions doesn't matter.

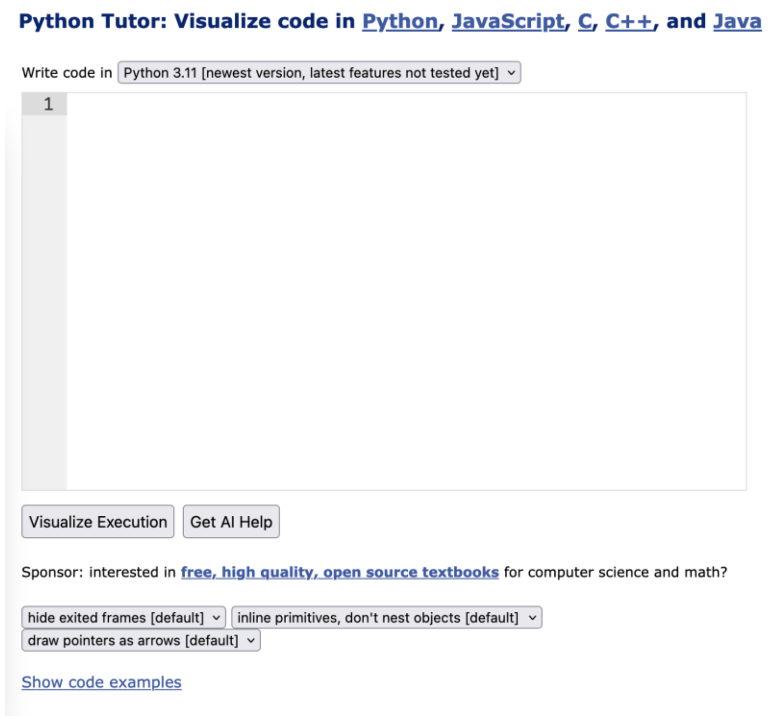
```
1.  main()
2.
3.  def main():
4.      print("I'm in main")
```

```
1.  def main():
2.      print_welcome()
3.
4.  def print_welcome():
5.      print("I'm in main")
6.
7.  main()
```

Visualizing And Debugging Functions

Python Tutor Can Help Visualize Interactions

- https://pythontutor.com/visualize.html#mode=edit
- Choose the latest version of Python available (3.11 at the time of writing)
- Paste the code in the following slide into Python Tutor, and then click the *Visualize Execution* button followed by the *Next* and *Prev* controls to step through the program.



Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

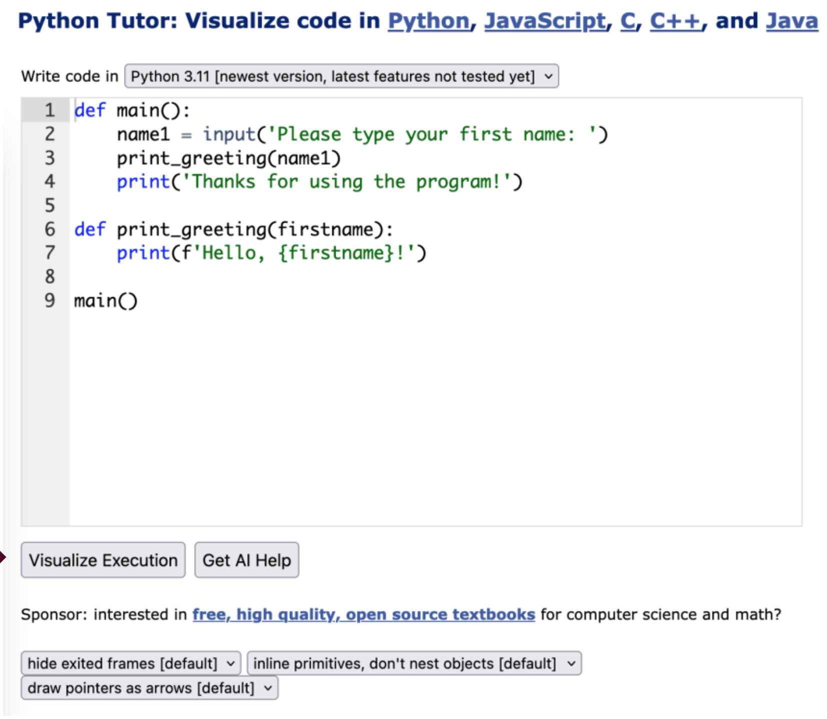
Write code in Python 3.11 [newest version, latest features not tested yet]

Sponsor: interested in free, high quality, open source textbooks for computer science and math?

hide exited frames [default]
inline primitives, don't nest objects [default]
draw pointers as arrows [default]

Python Tutor Can Help Visualize Interactions (cont.)

```python
def main():
    name1 = input('Please type your first name: ')
    print_greeting(name1)
    print('Thanks for using the program!')

def print_greeting(firstname):
    print(f'Hello, {firstname}!')

main()
```



Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

Write code in Python 3.11 [newest version, latest features not tested yet]

```python
def main():
    name1 = input('Please type your first name: ')
    print_greeting(name1)
    print('Thanks for using the program!')

def print_greeting(firstname):
    print(f'Hello, {firstname}!')

main()
```

Sponsor: interested in free, high quality, open source textbooks for computer science and math?

hide exited frames [default]
inline primitives, don't nest objects [default]
draw pointers as arrows [default]

Variable Scopes

WE BRIEFLY TALKED ABOUT BLOCKS AND SCOPES TWO WEEKS AGO.

SCOPE IS EVEN MORE CRITICAL WITH FUNCTIONS, SO READ CAREFULLY!

Scope and Local Variables

- Variables declared in a block, whether that’s a loop, an if/else block, or a function only exist until the block ends.
- **If you need to access data created in a function, you need to return that data and assign the result to a variable outside the function**
- This seems annoying, but it's a deliberate design choice common to almost all program languages.
- You don't need to use unique variable names throughout your program; they just need to be unique within a function.
- Data is contained within a function, so it is safe from other parts of your program
  - Other parts of your program can't create errors by accidentally modifying data in another scope
  - If you send data in and out of functions using parameters and return values, it's much safer

Scope Of Variables

```
def main():
    length = int(input('Enter length: '))
    width = int(input('Enter width: '))
    height = int(input('Enter height: '))
    calculate_volume()

def calculate_volume ():
    volume = length * width * height
    print('The volume of the box is: ', volume)

main()
```

This DOESN'T work as is – the calculate_volume function can't access variables from the main function. They are LOCAL to main.

Exception Handling

HOW TO HANDLE ERRORS GRACEFULLY

Crashes Vs User Error Messages

- When you run a program and it crashes, you will see a "traceback" message, which is very technical and contains information to help programmers understand what went wrong so they can fix it.
- Users can only fix their input and can't fix the program, so they should be told what *they* did wrong, not what *the program* did wrong.
- In a crash, the program did something wrong (like not safely handling user input)
- An error message just tells the user they did something wrong without the program crashing.
- From now on, our programs should *never* crash—they should only show error messages!

Exception Handling: `try` and `except`

- To handle errors, put the code that might cause an error in a `try` block.
- A `try` block is always paired with an `except` block.
- If an error happens in the `try` block, the code immediately jumps to the `except` block and runs whatever is inside that block. (This is called **throwing** and **catching** an exception—the `try` throws and the `except` catches.)

```
try:
    value = int(input("Enter a number: "))
except:
    print("Please only enter one or more digits 0-9.")
```

Exception Handling – Full Example

```python
value = None
while True:
    try:
        value = int(input("Enter a number: "))
        break
    except Exception as err:
        print(err.with_traceback) # Useful for debugging, but don't leave in to show the user
        print("Please only enter one or more digits 0-9.")
print(f'You entered the number {value}.')
```

Understanding Exceptions in Traceback Messages 🤔

Deep dive alert!

- So far, we've looked at generic exception handline, but we can handle specific types of exceptions in our code.
- First, we need to know what type of exception might be thrown:
  - If you try to run `my_value = 42/0`
  - You receive a traceback like this:

```
Traceback (most recent call last):
  File "c:\Users\erikg\src\demo.py", line 33, in <module>
    print(42 / 0)
          ~~~^~~
ZeroDivisionError: division by zero
```

- `ZeroDivisionError` is the specific error that was thrown.

Catching Specific Exceptions 🤔

- When we know what exceptions may be thrown, we can catch them individually:

```python
try:
    value = int(input("Please enter a number other than zero:"))
    print(f'42 divided by your value is {42 / value}.')
except ZeroDivisionError:
    print("Your entry must not be zero!")
except ValueError:
    print('You may only enter digits and optionally, a minus sign.')
except Exception:
    print('An unexpected error occurred.')
```

- It's a good idea to end a series of `excepts` like this with `Exception`, to handle any exceptions you didn't expect.

MIPO

Definition and Example

- MIPO:
  - Main
  - Inputs
  - Processing
  - Outputs
- Unless otherwise specified, you must structure your program with these four functions
- You should not have any code outside functions except for a single call to `main()` at the bottom of your file!
- Add additional functions as necessary, but the MIPO functions must exist.

MIPO Guidance

- The `main` function will control the flow of the program between the other functions.
- The `inputs` function is where the core user inputs are gathered—note that it is OK to get limited user input in other functions, but the key pieces need to be in `inputs`!
- The `processing` function is where calculations are performed
- The `outputs` function is where the main output is performed. Again, you can print small messages in other functions, but the main output needs to happen in this function.
- The program file `mipo_ex.py` is available for everyone in this week's module.
- We'll finish this week by reviewing the file and looking at its features.

Other Basic Features for the MIPO Model

- Starting a program over
  - At the end of the `main()` function, we want to ask the user if they would like to start over. If yes, call `main()` again. Else – say goodbye!
- Validating for integers
  - If you define a function to validate positive integers, you can call it as many times as you wish and use its return value to set variables in other parts of your program.