CS 134 Lecture 3: Functions
Check-in After First Lab!

- You have all survived your first computer science lab session
  - Congratulations!
- Software tools that you used:
  - **VS Code** as a text editor for code
  - **Terminal** as a text-based interface to the computer
  - **Git** for retrieving & submitting your work
  - **Python**, of course!

Do You Have Any Questions?
Announcements & Logistics

• **Lab 1**
  • Due today at 10 pm (for Mon labs), tomorrow at 10 pm (for Tues labs)
  • How to submit: make sure your work is up-to-date on evolene.cs.williams.edu

• **HW 2** will be released today, due next Monday at 10 pm
  • Open book/notes/computer. There is no time limit.

• **Optional** Personal machine setup (Mac/Windows): Step-by-step guide on website

• Lots of helps hours if you have questions!
  • Today noon-4 pm, 4-6 pm and 7-10 pm (in **TCL 216**)
  • Tomorrow 1-4 pm, 4-6 pm and 7-10 pm (in **TCL 216**)

---

**Do You Have Any Questions?**
Last Time

- Discussed **data types** and **variables** in Python
  - int, float, boolean, string
- Learned about basic **operators**
  - arithmetic, assignment
- Experimented with built-in Python functions
  - `input()`, `print()`, `int()`
- Discussed different ways to run and interact with Python
  - Create a file using an editor (VS Code), run as a script from Terminal
  - Interactively execute Python from Terminal
Today’s Plan

• Discuss functions in greater detail
• Review the built-in functions we (briefly) saw last time and in lab
  • `input()`, `print()`, `int()` all expect **argument(s)** within the parens
  • We will examine these a bit more today
• Learn how to define our own functions
Jupyter Notebook

- Last class we did examples in interactive python
- Upsides: low overhead, easy to use, can explore as you go
- Downsides:
  - No record of what we did
  - Can't pre-type examples to run in class
- For today, we will try using Jupyter Notebook for lecture examples
  - Jupyter notebook is an “enhanced” way to use interactive python
  - Installed on lab machines & included in personal machine setup guide
- Anything we do in Jupyter notebook can be done in Interactive Python!
- Regardless of format, all examples will be posted on the website
Review: Python Built-in Functions

input(), print()
int(), float(), str()
Built-in functions: input()

- **input()** displays its single argument as a prompt on the screen and waits for the user to input text, followed by **Enter/Return**
- It interprets the entered value as a **string** (a sequence of characters)

```python
>>> input('Enter your name: ')  Enter your name: Charlie Brown  'Charlie Brown'
>>> age = input('Enter your age: ')  Enter your age: 8
>>> age
'8'
```

Prompts in Maroon. User input in blue. Inputted values are by default a **string**
Built-in functions: print()

- `print()` displays a character-based representation of its argument(s) on the screen/Terminal.

```python
>>> name = 'Peppermint Patty'
>>> print('Your name is', name)
Your name is Peppermint Patty
>>> age = input('Enter your age : ')  
Enter your age: 7
>>> print('The age of ' + name + ' is ' + age)
The age of Peppermint Patty is 7
```
Built-in functions: int()

When given a string that's a sequence of digits, optionally preceded by + or -, int() returns the corresponding integer

- On any other string, int() raises a ValueError
- When given a float, int() returns the integer that results after truncating the fractional part (rounds towards zero)
- When given an integer, int() returns that same integer

```python
>>> int('42')
42
>>> int('-5')
-5
>>> int('3.141')
ValueError
```
Built-in functions: float()

When given a string that’s a sequence of digits, optionally preceded by + or -, and optionally including one decimal point, float() returns the corresponding floating point number:

- On any other string float() raises a ValueError
- When given an integer, float() converts it to a floating point number.
- When given a floating point number, float returns that number

```python
>>> float('3.141')
3.141
>>> float('-273.15')
-273.15
>>> float('3.1.4')
ValueError
```
Built-in functions: str()

- Converts a given type to a string and returns it
- Returns a syntax error when given invalid input

```python
>>> str(3.141)
'3.141'
>>> str(None)
'None'
>>> str(134)
'134'
>>> str($)
SyntaxError: invalid syntax
```
Today:
User-Defined Functions
Organizing Code with Functions

• So far we have:
  • Written simple *expressions* in Python
  • Created small scripts to perform concrete tasks
• This is fine for small computations!
• Need more organization and structure for larger problems
• Structured code is good for:
  • Keeping track of which part of our code is doing what actions
  • Keeping track of what information needs to supplied where
  • **Reusability!** Specifically, reusing blocks of code
Abstracting with Functions

- **Abstraction**: Reduce code complexity by ignoring (or hiding) some implementations details
  - Allows us to **decompose** and **reuse** parts of our code
- **Real life example**: a video projector
  - We know how to switch it on and off (**public interface**)  
  - We know how to connect it to our computer (**input/output**)  
  - We don’t know how it works internally (**information hiding**)  
- **Key idea**: We don't need to know much about the internals of a projector to be able to use it  
  - Same is true with **functions**!
Decomposition

• Divide **individual tasks** in our code into **separate functions**
  • Functions are **self-contained** and **reusable**
  • Each function is a **small piece** of a **larger task**
  • Keeps code **organized** and **coherent**
• We have already seen some built-in examples (**int()**, **input()**, **print()**, etc.)
• Now we will learn how to **decompose** our Python code and hide small details using **user-defined functions**
• Later we will learn a new abstraction which achieves a greater level of decomposition and information hiding: **classes**
Anatomy of a Function

- Function **definition** characteristics:
  - Has a **header** consisting of:
    - **name** of the function
    - **parameters** (optional)
    - **docstring** (optional, but strongly recommended)
  - Has a **body** (indented and required)
  - Always **returns** something (with or without an explicit **return** statement)
  - **Statements** within the body of a function are not run in a program until they are “called” or “invoked” through a **function call** (like calling `print()` or `int()` in your program)
**Function Example**

**Function definition**

```python
def square(x):
    '''Takes a number x and returns its square'''
    return x*x
```

**Function Calls/Invocations**

```python
>>> square(5)
25

>>> square(-2)
4
```
Function Example

**Function definition**

def square(x):
    '''Takes a number x and returns its square'''
    return x*x

**Function Calls/Invocations**

>>> square(5)
25

>>> square(-2)
4
Function Example

**Function definition**

```python
def square(x):
    '''Takes a number x and returns its square'''
    return x*x
```

**Function Calls/Invocations**

```python
>>> square(5)
25
>>> square(-2)
4
```
Function Example

**Function definition**

```python
def square(x):
    '''Takes a number x and returns its square'''
    return x*x
```

**Function Calls/Invocations**

```python
>>> square(5)
25

>>> square(-2)
4
```
Function Example

**Function definition**

def square(x):
    '''Takes a number and returns its square'''
    return x**x

This is the body of the function. Notice the use of an explicit `return` statement.

**Function Calls/Invocations**

```python
>>> square(5)
25
```

```python
>>> square(-2)
4
```
**Function Example**

**Function definition**

def square(x):
    '''Takes a number and returns its square'''
    return x*x

**Function Calls/Invocations**

```python
>>> square(5)
25

>>> square(-2)
4
```

When we call/invoke the function, 5 is the **argument** value. Function is evaluated using x=5.
**Function Example**

**Function definition**

```python
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

**Function Calls/Invocations**

```python
>>> square(5)
25
>>> square(-2)
4
```

**Summary:**
- Indent in function body (required)
- Colon after function name (required)
- Docstring (recommended, good style)
- `x` in function definition is a parameter
- Single line body which returns the result of the expression `x * x`
- `return` always ends execution!
- A function is defined once and can be called any number of times!
A Closer Look At Parameters

• **Parameters** are “placeholders” in the body of a function that will be filled in with **argument values** during each invocation.

• A particular name for a parameter is irrelevant, as long as we use it consistently in the body (just like \( f(x) \) and \( f(y) \) in math).

  • All **square** function definitions below work exactly the same way!
  
  • Invocation would also look exactly the same: `square(5)`

```
def square(x):
    return x*x

def square(num):
    return num*num

def square(apple):
    return apple*apple
```

**Rule of thumb:** Choose parameter names that make sense and convey meaning.
Python Function Call Model

**Function frame:** Model for understanding how a function call works

```python
def square(x):
    return x * x
```

- `square(2+3)`: 5
- `square(5)`: 5
- `square(25)`: 25

*Return value replaces the function call!*
Function Call Replaced by Return Value

17 + \text{square}(2+3)

17 + \text{square}(5)

17 + 25

42
Print() vs Functions that Return Values

- Notice that the `print()` function does not `return` any value:
  - No `Out[]` cell when we print in Jupyter

- In contrast to `print()`:
  - `input()` function returns the value inputted by user as a str
  - `int()` function returns the given value as type `int`
  - `type()` function returns the type of given value, etc

- Functions that do not explicitly return a value, implicitly return `None`
Value vs. None Returning Functions

We call functions that return a None value **None-returning functions**. Such functions are invoked to perform an action (e.g., print something, change state). They do **not compute and return a result**.

We call functions that return a value other than None **value returning functions**.

**Value Returning**

```python
def square(x):
    return x**x
```

**None-Returning**

```python
def printHW():
    print('Hello World')
```

What if I run `print(printHW)` or `print(print((printHW)))`?
Return Statements

- **return** only has meaning *inside* of a function definition

- A function definition may have multiple returns, but only the first one encountered is executed!

- Any code that exists after a return statement is unreachable and will not be executed

- The value returned by the function’s return statement replaces the function call in a computation

- Functions without an explicit return statement implicitly return **None**