CSI 34:
Functions
Check-in After First Lab!

• You have all survived your first computer science lab
  • Congratulations!
• Computer science 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?
Aside: Submitting Labs via Git

- Git is a version control system that lets you manage and keep track of your source code history
- Key commands:
  - `git clone` - every time you start a new lab OR move to a new machines, use `git clone` to download the latest copy of your code from our server
  - `git add <files>` - mark <files> to be uploaded to server on next push
  - `git commit -m "message"` - create a checkpoint, used after `git add`
  - `git commit -am "message"` - combines add and commit into one step; *only use for files that have been previously added*
  - `git push` - send files that were added/committed to server
  - `git pull` - get latest code from server (after you have cloned)
Aside: Useful Unix Commands

- **pwd** - print working directory
- **mkdir <dir name>** - make new directory (or folder)
- **cd <dir name>** - change directory

  - Special directory names
    - . (single dot, current directory)
    - .. (two dots, parent directory)
    - ~ (tilde, home directory)

- **cd ..** takes you to the parent directory
- **cd** takes you “home”
- **ls** shows contents of current directory
Announcements & Logistics

• **Lab 1**
  • Due today at 10 pm (for Monday labs)
  • Due tomorrow at 10 pm (for Tuesday labs)
  • Make sure your work has been added/committed/pushed to evolene using git

• **Homework 2** released today on Glow, due next Monday at 10 pm
  • Open book/notes/computer
  • No time limit

• **Student help hours and TA hours - check calendar**
  • If you are in isolation and need to chat, let us know! We’ll set up a time to Zoom

---

Do You Have Any Questions?
Aside: Jupyter Notebooks

• You can experiment with examples that we do in class using our **Jupyter notebooks**
  
• Jupyter notebooks often contain additional examples beyond what we cover in lecture
  
• For extra practice, we recommend running these examples on your own (using Jupyter or in Interactive Python)
  
• Reviewing these notebooks is also a great way to review lecture material and study for exams
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 (or Jupyter notebook)
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
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: Marcel the Shell  
'Marcel the Shell'
>>> age = input('Enter your age: ')  
Enter your age: 12
>>> age
'12'
```

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 = 'Marcel the Shell'
>>> print('Your name is', name)
Your name is Marcel the Shell
>>> age = input('Enter your age : ')
Enter your age: 12
>>> print('The age of ' + name + ' is ' + age)
The age of Marcel the Shell is 12
```

Comma as a separator adds a space

Can also add spaces through string *concatenation*
Built-in functions: int()

- When given a string that's a sequence of digits, optionally preceded by +/-, `int()` returns the corresponding integer.
- On any other string it raises a `ValueError`.
- When given a float, `int()` returns the integer that results after truncating it 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 +/-, and optionally including one decimal point, float() returns the corresponding floating point number.
- On any other string it 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 certain 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 achieve code *decomposition* and reuse

- **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
  - Keep code organized and coherent
- We have already seen some built-in examples (\texttt{int()}, \texttt{input()}, \texttt{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 code hiding: classes
Anatomy of a Function

• Function definition characteristics:

• A header consisting of:
  • name of the function
  • parameters (optional)
  • docstring (optional, but strongly recommended)

• 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 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 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

Function Calls/Invocations

>>> square(5)
25

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

**Function definition**

```python
def square(x):
    '''Takes a number and returns its square'''
    return x**2
```

**Function Calls/Invocations**

```plaintext
>>> 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

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

Function Calls/Invocations

>>> 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!
- Function is defined once and can be called any number of times!
A Closer Look At Parameters

- **Parameters** are “holes” in the body of a function that will be filled in with *argument values* in 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 of the `square` function definitions work exactly the same way!
  
  - Invocation would also look exactly the same: `square(5)`

```python
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. Avoid always using \( x \), for example.
**Python Function Call Model**

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

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

```
square (2+3) → square (5) → 25
```

Return value replaces the function call!

```
square frame
x= 5
return x * x
```

```
square frame
x= 5
return 5 * 5
```

```
square frame
x= 5
return 25
```
Function Call Replaced by Return Value

\[ 17 + \text{square}(2+3) \]
\[ = 17 + \text{square}(5) \]
\[ = 17 + 25 \]
\[ = 42 \]
Interactive Python:
Let’s See Some Examples