CS134: Functions

Check-in After First Lab!

- You have all survived your first computer science lab
  - **Congratulations!**
- Computer science tools that you used:
  - **Atom** as a text editor
  - **Terminal** as a text-based interface to the computer
  - **Git** for versioning, **Gitlab** 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 <file> - mark <file> to be uploaded to server
  - git commit -m "message" - create a checkpoint
  - git commit -am "message" - combines add and commit into one step; only use for files that have been previously added
  - git push - send code to server
  - git pull - get latest code from server (after you have cl

Aside: Directories in Unix

- 'Folders' on your computers are called 'directories' in Unix-based operating systems
- Your 'current directory' is important when executing commands on the Terminal
  - For example, programs that run as a script, such as `helloworld.py`, must be in the *same* directory as where you execute the command `python3 helloworld.py`
  - Otherwise your computer doesn't know which program to run
- Similarly, when you `git pull`, you need to be in the correct directory
- Useful to learn how to navigate between directories with the Terminal

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 11 pm (for Monday labs)
  - Due tomorrow at 11 pm (for Tuesday labs)
  - Just make sure your work has been commit/pushed to evolene using git
- **Homework 2** released today on Glow, due next Monday at 11 pm
  - Open book/notes/computer
  - No time limit
- **Office hours and TA hours - check calendar**
  - Zoom today
  - In person starting tomorrow! :-)

Do You Have Any Questions?

Aside: Accessing Lecture Materials

CSCI 134: Introduction to Computer Science

Search this site...

- Calendar
- Syllabus
- Office and TA Hours
- Resources
- Computer Science Department
- Williams College

Scroll to Today

MON	TUE	WED	THU	FRI
01/31	02/01	02/02	02/03	02/04 **Welcome & Logistics** • Read Ch 1 • Slides • **HW 1** • **Lab 1**
02/07 **Types & Expressions** • Read Ch 2 • Notebook • Slides • **DUE: HW 1**	02/08	02/09 **Functions**	02/10	02/11 **Booleans & Conditionals**

Aside: Jupyter Notebooks

- How can you experiment with examples that we do in class with a **Jupyter notebook** by yourself?
  - Jupyter notebooks often contain examples beyond what we cover in lecture
  - For extra practice, we recommend running these examples on your own
  - Reviewing these also notebooks is a great way to review lecture material and study for exams
  - Instructions for installing and running Jupyter available on webpage
  - Come find us during office hours if you need help!

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
  - `int(), input(), print()`
- Discussed different ways to run and interact with Python
  - Create a file using an editor (Atom), run as a script from Terminal
  - Interactively execute Python from Terminal (or Jupyter notebook)

Today

- 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
- Discuss the distinction between **fruitful** and **non-fruitful** functions
- Learn how to define our own functions

Review:
Python Built-in Functions

`int(), float(), str()`

`input(), print()`

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

```
>>> int('42')
42
>>> int(-5.5)
-5
>>> int('3.141')
ValueError
```

Interactive Python
(>>> indicates Python prompt)

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

```
>>> 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

```
>>> str(3.141)
'3.141'
>>> str(None)
'None'
>>> str(134)
'134'
>>> str($)
SyntaxError: invalid syntax
```

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 returns the entered value as a **string**

```
>>> input('Enter your name: ')
Enter your name: Harry Potter
'Harry Potter'
>>> age = input('Enter your age : ')
Enter your age: 17
>>> age
'17'
```

Prompts in red. 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 and returns a special **`None`** value (not displayed). Notice there are no "Out[]" lines.

Comma as a separator adds a space

```
>>> name = 'Harry Potter'
>>> print('Your name is', name)
Your name is Harry Potter
>>> age = input('Enter your age : ')
Enter your age: 17
>>> print('The age of ' + name + ' is ' + age)
The age of Harry Potter is 17
```

Can also add spaces through string *concatenation*

Today:
User-Defined Functions

Structuring Code

- So far we have:
  - Written simple **expressions** in Python
  - Created small scripts to perform certain tasks
- This is fine for small computations!
  - But we need more organization for larger problems
- Structuring 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

- To write organized code, we divide individual tasks into separate functions
    - Functions are **self-contained**
    - Each function is a **small piece** of a **larger task**
    - Functions are **reusable**
    - Keep code **organized**
    - Keep code **coherent**
- We have already seen some built-in examples (`int(), input(), print()`, etc)
- Today we will learn how to **decompose** our Python code and hide small details using **user-defined functions**
- Later in the semester, 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 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

Function's name is `square`

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

Function Calls/Invocations

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

Function Example

`square` has one **parameter**, `x`, which is the expected input to the function.

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

This is the **docstring**, which is enclosed in triple quotes. It is a short description of the function.

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

This is the body of the function. Notice that this functions includes an explicit **return** statement.

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

Notice the indentation. This is very important!!

```
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

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

When we call/invoke the function, 5 is the **argument** value.

Function Calls/Invocations

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

Function Example

Function definition

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

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!

Function Calls/Invocations

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

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)`

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

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

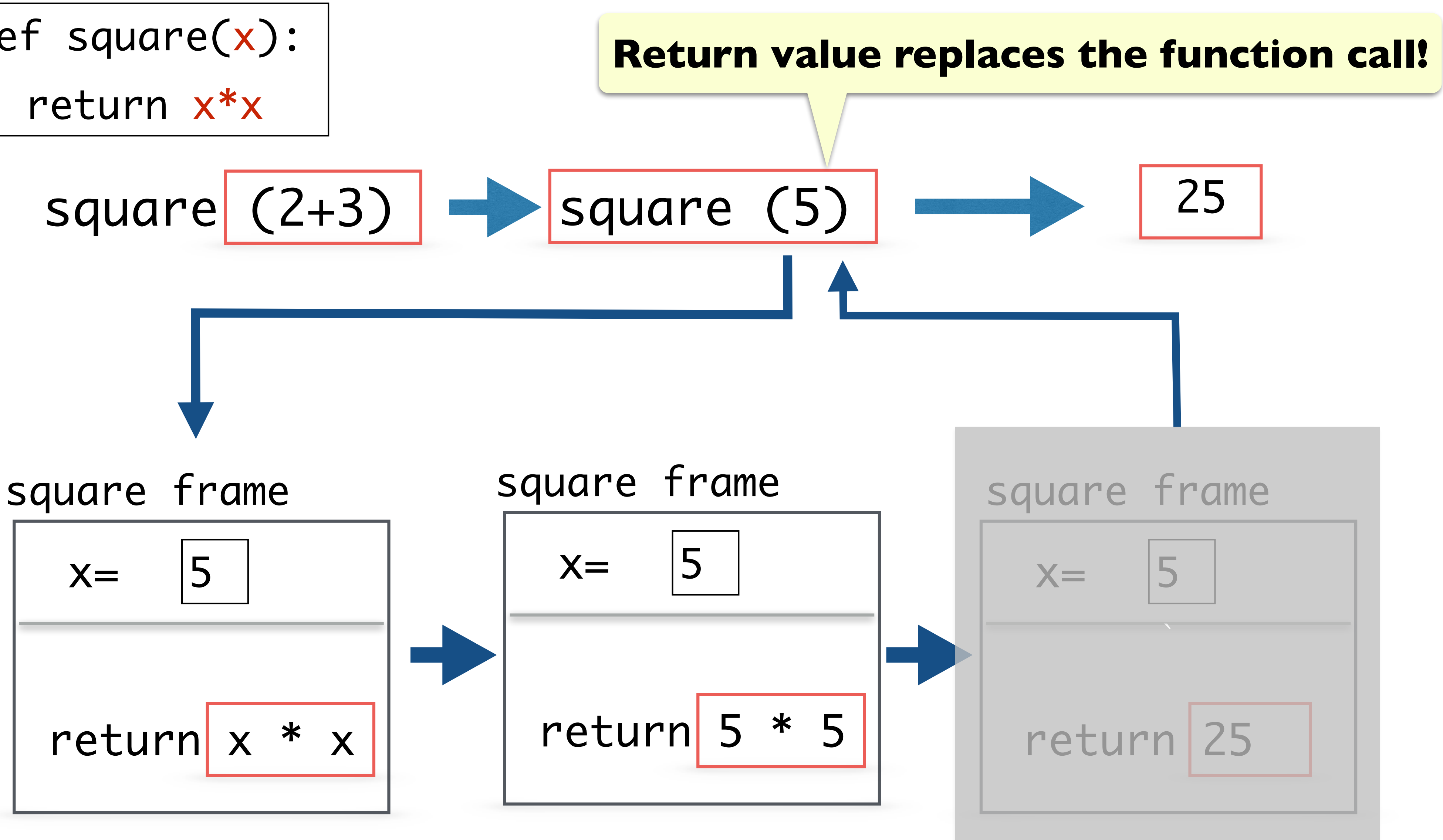
```
def square(num):
  return num*num
```

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

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



Function Call Replaced by Return Value

```
17 + square (2+3)

17 + square (5)

17 + 25

42
```

Jupyter Notebook:
Let's See Some Examples