CSCI 134 Lecture 2:
Python Types and Expressions

Announcements & Logistics

- **HW 1** due today at 10 pm (Google form)
- **Lab 1** today/tomorrow, due Wed/Thur at 10pm
  - Gain experience with the workflow and tools
  - Start with some short and sweet Python programs
  - *Important*: Login to Lab machines using **OIT credentials**
  - `clone/pull/push` to evolene.cs.williams.edu with **CS credentials**
    - You must have received an email about CS account info!
- **Student help hours and TA hours have started**
  - Check calendar on course webpage
- **Questions?**

Last Time

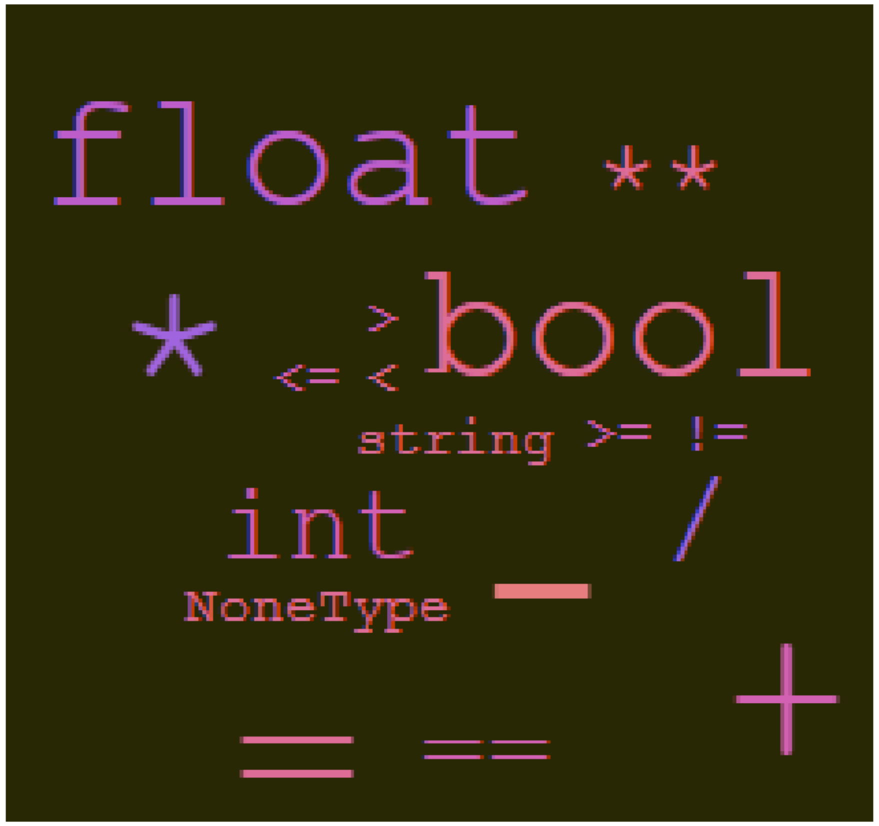
- Discussed course logistics
- **Reviewed** syllabus
- Important take-aways:
  - cs134 course website: place where everything is hosted
- Encouraged to use lab machines but resources to setup your personal machines are available on the website
  - Reach out to us or TAs if you get stuck

Today’s Plan

- Learn lots of new vocabulary words!
- Discuss **data types** and **variables** in Python
  - `int, float, boolean, string`
- Learn about basic **operators**
  - arithmetic, assignment
- Experiment with built-in Python **functions** and expressions
  - `int(), input(), print()`
- Investigate different ways to run and interact with Python

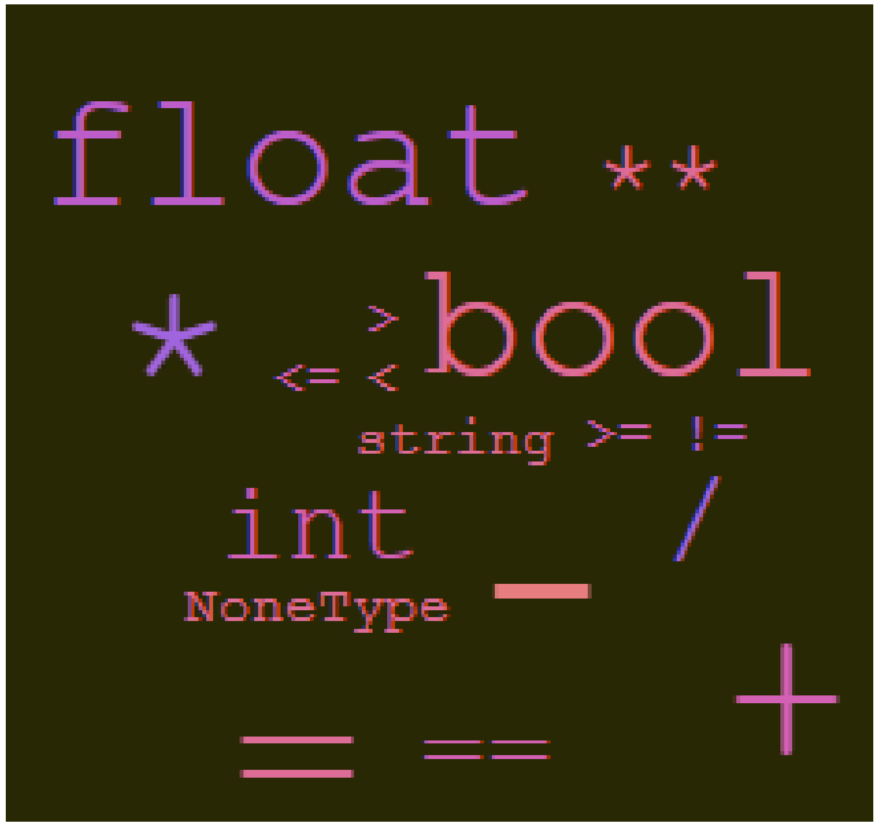
Aspects of Languages

- **Primitive constructs**
  - English:
    - words, punctuation
  - Programming languages:
    - numbers, strings, simple operators



Aspects of Languages

- **Syntax**
  - English:
    - “boy dog cat” (incorrect), “boy hugs cat” (correct)
    - “Let’s eat grandma!” (probably incorrect), “Let’s eat, grandma!” (correct)
  - Programming language:
    - `“hi”5` (incorrect), `4*5` (correct)



Aspects of Languages

- **Semantics** is the meaning associated with a syntactically correct string of symbols
  - **English:**
    - Can have many meanings (ambiguous), e.g.
    - “Flying planes can be dangerous”
    - Other examples?
  - **Programming languages:**
    - Must be *un*ambiguous
    - Can only have one meaning
    - Actual behavior is not always the intended behavior!

Python3

- Programming language used in this course
- Great introductory language
    - Better human readability and user friendly syntax than other PLs
- For this class, we need `Python 3.10`
- Checking version of Python on machine
    - Type `python3 --version` in Terminal (VS Code Terminal for Windows)
- **Preinstalled on all lab machines**
- Installing Python3 on your machine: see setup guide

Python Interfaces

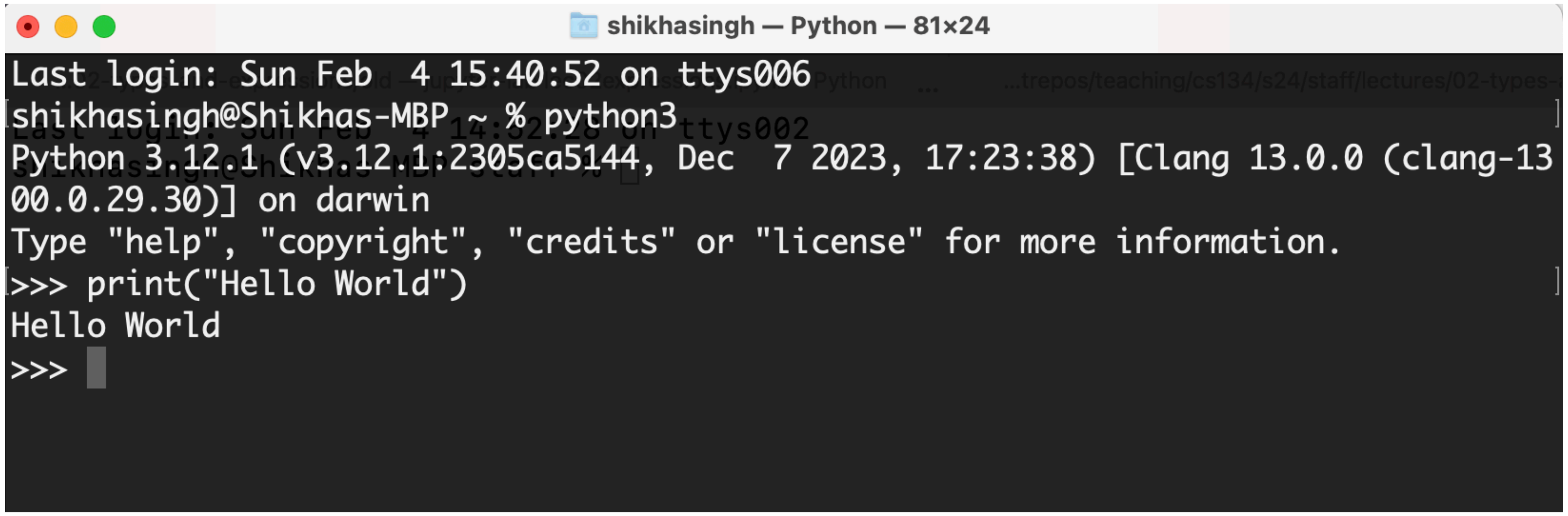
- You can run Python code in two ways:
    - As a **script**
        - Save code in a file, run from Terminal
    - **Interactively** (from Terminal)
        - Interactive session

Python: Program as a Script

- A **program** is a sequence of definitions and commands
  - Definitions are evaluated
  - Commands are executed and instruct the interpreter to do something
- Type instructions in a **file** that is read and evaluated sequentially
  - e.g., last lecture we wrote **`helloworld.py`** in a file and then executed it from the Terminal with
    `python3 helloworld.py`
  - **Standard method**: good for longer pieces of code or programs
  - We will use this method in our labs
  - Called "running the Python program as a *script*”

Python: Interactive

- Running Python **interactively** is great for introductory programming
- Launch the Python interpreter by typing **`python3`** in the Terminal
    - Opens up Interactive Python
    - Almost like a "calculator" for Python commands
    - Takes a Python expression as input and spits out the results of the expression as output
    - Great for trying out short pieces of code



```
Last login: Sun Feb  4 15:40:52 on ttys006
shikhasingh@Shikhas-MBP ~ % python3
Python 3.12.1 (v3.12.1:2305ca5144, Dec  7 2023, 17:23:38) [Clang 13.0.0 (clang-13
00.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 
```

Python Primitive Types

- Every data **value** has a data **type**. For example:
  - `10` is an integer (type: **int**)
  - `3.145` is a decimal number (type: **float**)
  - `'Williams'` or `"Williams"` is a sequence of characters (type: **string**)

Knowing the **type** of a **value** allows us to choose the right ***operator*** for expressions.

Python Primitive Types

- Every data **value** has a data **type**. For example:
  - `10` is an integer (type: **int**)
  - `3.145` is a decimal number (type: **float**)
  - ‘`Williams`’ or “`Williams`” is a sequence of characters (type: **string**)
  - `0` (`False`) and `1` (`True`) (type: **boolean** or **bool**)
    - Represent answers to decision questions (yes/no)
  - *Empty value* (type: **None**)
- We will revisit booleans and None types soon!

Knowing the **type** of a **value** allows us to choose the right ***operator*** for expressions.

>>> examples

Python Operators

- **Arithmetic operators:**
  - **+** (addition), **-** (subtraction), ***** (multiplication)
  - **/** (floating point division, returns a value with a decimal point)
  - **//** (integer division, returns an integer)
  - **%** (modulo, or remainder)
  - ****** (power, or exponent)
- **Assignment operator:**
  - **=** ("is assigned or gets", not "equals")
  - Used to "assign" values to **variables**
  - ***Note.*** Not to be confused with mathematical equality, which is written as **==** in programming languages

Variables & Assignment

Variables and Assignments

- A **variable** names a value that we want to use later in a program
    - If we define `num = 17` then the value `17` essentially gets stored in a slot in memory with the label `num`
    - We are **assigning** `num` (a variable) the value `17`

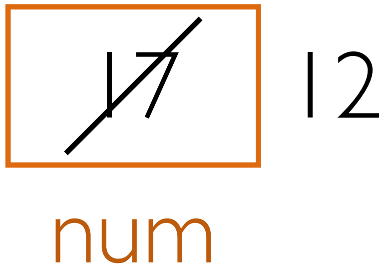
- Once defined, we can reuse variable names again, and later assignments can change the value in a variable box
    - `num = num - 5`
    - What is stored in `num` after this evaluates?

Math vs Programming. An assignment: expression on the right evaluated first and the value is stored in the variable name on the left

Variables and Assignments

- A **variable** names a value that we want to use later in a program
    - If we define `num = 17` then the value `17` essentially gets stored in a slot in memory with the label `num`
    - We are **assigning** `num` (a variable) the value `17`

- Once defined, we can reuse variable names again, and later assignments can change the value in a variable box
    - `num = num - 5`
    - What is stored in `num` after this evaluates?
    - var = <expression> (result of expression gets stored in the variable box var)



- **Question**. *Why would we want to name values or expressions?*

Abstracting Expressions

- Why give names to data values or the results of expressions?
  - To **reuse** names instead of values
  - Easier to change code later
- For example:

```
pi = 3.1415926  # useful to name
radius = 2.2
area = pi * (radius**2)
# suppose now we want to change radius
radius = 2.2 + 1
area = pi * (radius**2) # new area
```

Python Built-In Functions

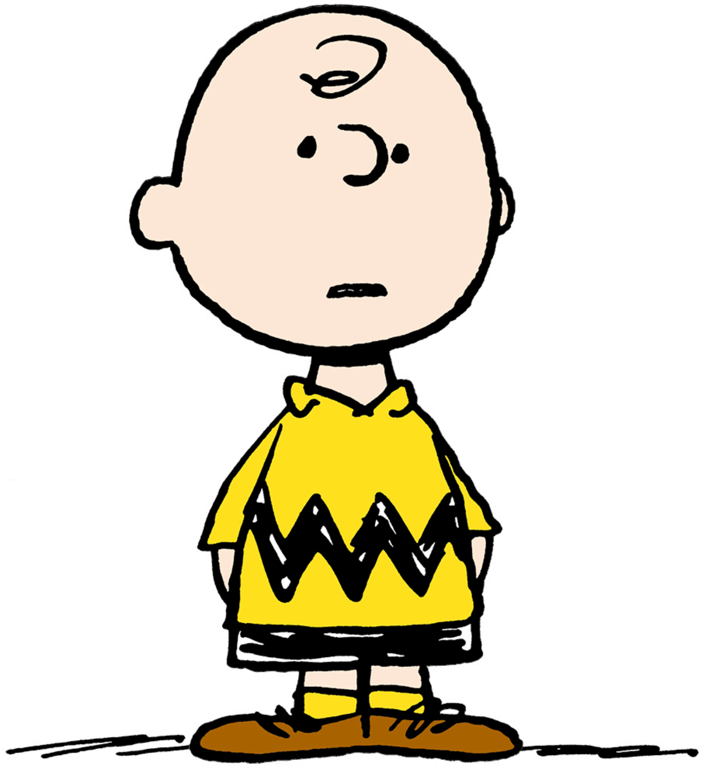
Built-In Functions

- Python comes with a ton of built-in capabilities in the form of **functions**
  - We will discuss the following built-in functions today
  - `input(), print()`
  - `int(), float(), str()`
- Will formally discuss functions on Friday

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**
- ***Important***: interprets the entered value as a **string**

```
>>> 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.

```
>>> 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
```

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

```
>>> 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

```
>>> 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
```

[Aside] Comments and Indenting

- Anything after **#** in Python is a comment
  - Ignored by the interpreter
  - Meant for humans reading the code
  - Useful for readability for large pieces of code
- Python is sensitive to **indentation**
  - Signify start of new "code block"
  - We will see how to use indents more in the coming lecture

>>> examples

Understanding Git

- Git is a version control system that lets you manage and keep track of your source code history
- **GitHub** is a cloud-based git repository management & hosting service
  - **Collaboration**: Lets you share your code with others, giving them power to make revisions or edits
- **GitLab** (on `evolene.cs.williams.edu`) is similar to GitHub but maintained internally at Williams
  - All your lab files "live" on the CS server
  - **Cloning** it creates a local copy that you can work on
  - commit/push lets you send updates to the local files to the server

Git Commands

- `clone`
  - creates a local copy of the repository on the server
- `status`
  - gives you the git status of all works in current directory
- `add`
  - "stages" the changes in a local file to be sent to the server
- `commit`
  - commits the "added" changes
- `push`
  - pushes the committed changes to the server

An 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

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