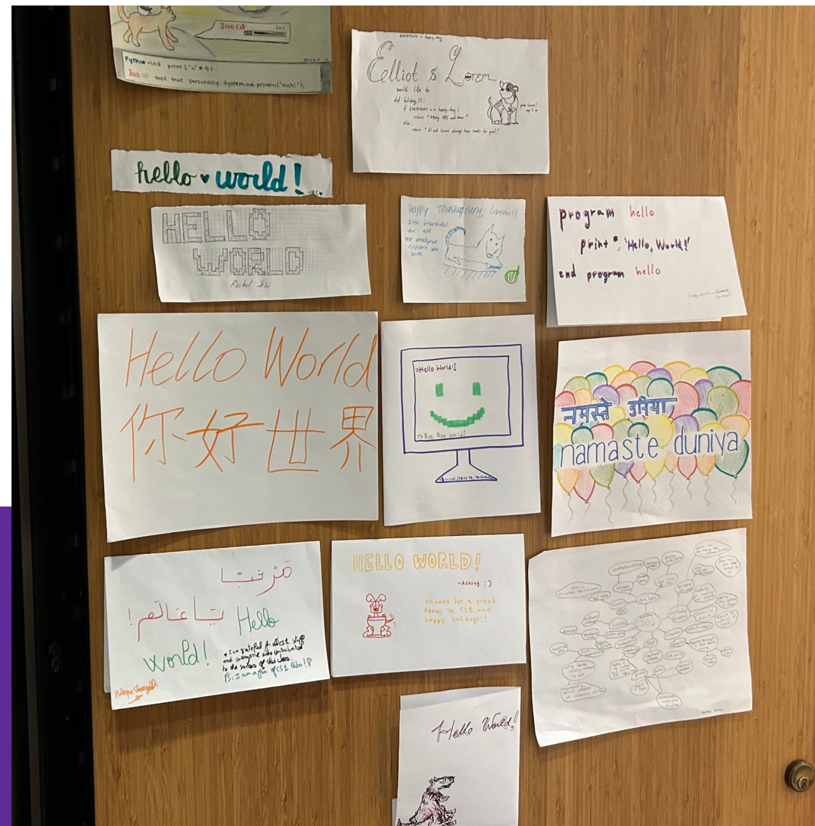
CS 1: Intro to CS

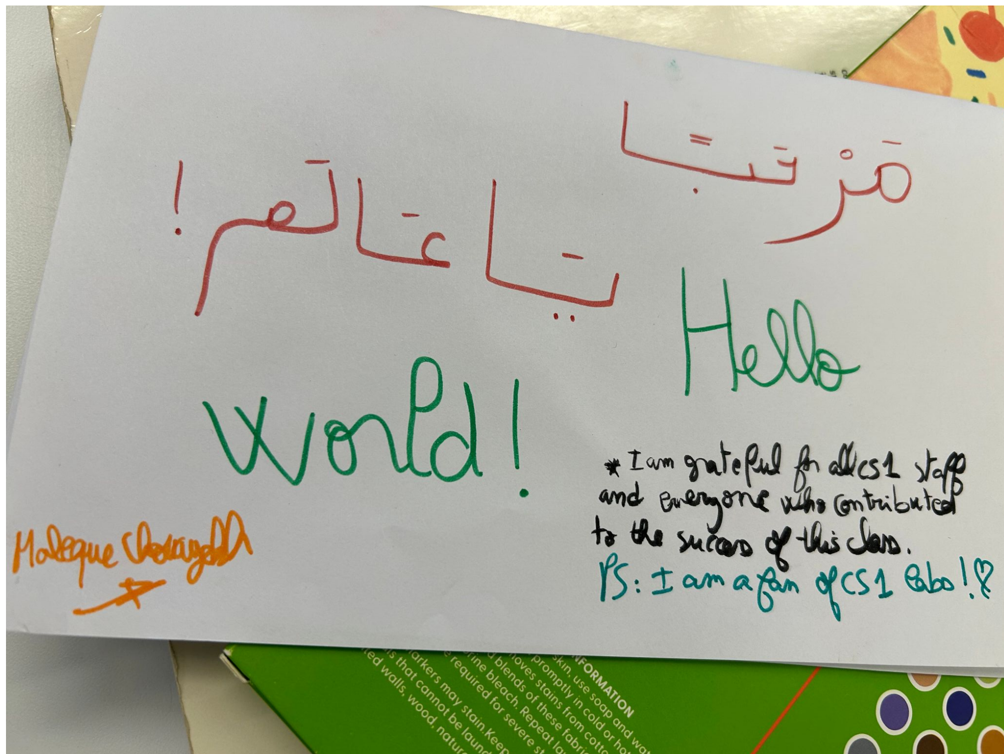
Lecture 1: Intro to CS 1 and Python



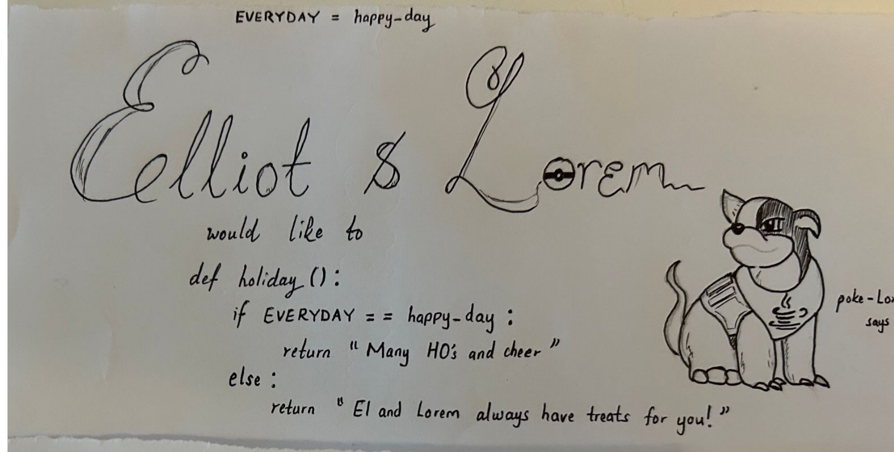
Monday, Apr. 1st

Agenda

- Introductions
- Course Overview
- Getting started with Python
  - Writing and executing
  - Types and expressions
  - Variables and assignment



CS 1 Staff



Instructor:

- El Hovik (hovik@caltech.edu) (they/she)
- 3rd year at Caltech, from Seattle, WA
- Switched from Pre-Medicine to CS
- I enjoy bridging theory and application in CS (one of the reasons I love teaching CS1, CS 121 Databases, and CS 132 Web Development!)
- In my free time, I enjoy puzzles, games, podcasts, learning new languages, mentoring, and spending time with my four-legged friend Lorem
- I have experience in PL and CS Education research, as well as interning as a SWE at Expedia and a few start-ups in Seattle

Course mascot: Lorem (Ipsum)

7 TAs! (Introductions)

My Teaching Philosophy

- I do not assume you have programming experience coming in (I took my first CS class in college!)
- Each student comes in with different backgrounds, interests, goals, life experiences
- Grades do not define you
- You are all learning how to be a college student, while also learning how to be an adult (and so are all of the TAs and faculty at Caltech!)
- Students have a range of learning styles, from visual, passive vs. active, auditory, oral, etc.
- You are all innately good at problem-solving; in CS 1, we will learn how to leverage this human skill to formalizing problems in a variety of real-world problems ranging from programming fundamentals, data science, Biology, Chemistry, Math, Art, Economics, Linguistics, Ethics, Privacy… the list goes on!
- CS 1 is not about learning Python; it's about learning problem-solving and buildings toolset that will be essential to your success in Caltech academics, research, internships, etc.
- I care more about students demonstrating understanding than meeting deadlines

Course Tools

Canvas: https://caltech.instructure.com/courses/4597

Course website: https://eipsum.github.io/cs1

VSCode: Python coding environment

- You may use bash or a different editor if you prefer, but **do not** use Jupiter notebooks or Anaconda

Discord: Zoom Office Hours and quick questions

CodePost: HW submission and feedback

Organization

Lectures: MWF 1-1:55PM

Tuesday Labs (1-hour, required attendance; times will be confirmed after Discord student poll - **please respond with a react by today!**)

Weekly HW "Mini Project" Sets (HW1 out this week, due next Thursday 11:30PM)

Final Exam

Course Objectives

- Write, document, test and debug programs of up to a few hundred lines of code in Python through a variety of interdisciplinary applications
- Understand the differences between a program and a programming language through a brief introduction to Java compared to Python
- Understand the fundamentals of imperative and object-oriented programming
- Develop resilience in coding by gaining familiarity with common errors and debugging strategies
- Fostering confidence in students' ability to troubleshoot code and fix errors

Course Schedule

Week	Topic	Labs	Assignments
1	Python Basics 1: variables, functions, strings	Lab 00*	-
2	Python Basics 2: debugging, documentation, lists	Lab 01	HW1
3	Control Structures: if statements, loops	Lab 02	MP2
4	Intro to File Processing: tuples, dictionaries	Lab 03	MP3
5	Data Manipulation: CSV processing	Lab 04	MP4
6	Data Visualization: Matplotlib	Lab 05	MP5
7	Intro to Object Oriented Programming (OOP)	Lab 06	MP6
8	Programming from Python to Java	Lab 07	MP7
9	Object Oriented Programming in Java	Lab 08	MP8
10	Victory Lap: final review	Lab 09*	-
11	-	-	Final

Lectures and Readings

Each lecture will have one or more readings posted to prepare for the new material

- You can find the first two readings here and here

Lectures will generally be interactive, involve coding and having slides mostly for reference

Questions are encouraged!

How to Use Lectures

I will be incorporating activities in most lectures; the best ones have been "on-the-fly" based on student questions, and these are shaped the more I get to know you all

My goal is to make lectures worth your time, focusing on live-coding and discussions of common pitfalls, HW tips, and trade-offs; feedback/requests are welcome!

Slides and readings are comprehensive, so **it is expected that you don't assume all slides will be covered in lecture**; that said, El will always incorporate the key material in lecture and summarize take-aways

Slides will be posted in advance, but occasionally will shift around (e.g. if we don't get through the last few due to an activity)

Lecture Checks

Optional exercises designed to support your studying and guide the readings as you learn new material (published as Canvas Quizzes)

You are encouraged to discuss the questions in OH or Discord, and *may* collaborate with other students

We do not expect you to spend more than 30-45 minutes on readings/Lecture Checks, but the time you do spend will be important to introduce the fundamentals so that lecture can reinforce what you read and provide motivation, demonstrate code examples, and discuss practical use-cases during lecture.

Attendance

Lecture attendance is optional...

That said, we are making lectures more interactive and focused on problem-solving strategies (with details of material covered in readings) that will help you prepare for assignments and CS beyond the course

You can also earn Rework Tokens with lecture exit tickets (short forms) to apply to reworking assignments (more details soon)

Lecture recordings will not be available to students unless there are approved exceptional cases

Grading Policy

- HW and MPs: 24pts (8×3pts)
- Labs: 7pts (7×1pt - lowest grade dropped)
- Final: 10pts

A total of 41pts can be obtained in the course and the passing line is set to 33pts. Additional points may be obtained through *engagement opportunities*; throughout the term, the instructor may offer students the opportunity to earn additional points by attending talks, writing blog posts, submitting TQFR... The lowest lab grade will be dropped when calculating final grades. The course staff reserves the right to modify the grading policy at any point throughout the course.

Grading Homework

Each assignment has multiple parts in rubric:

- Correctness
- Documentation
- Code Quality
- ...

Each assignment will be graded out of 30 before being divided by 10 to give a grade between 0 and 3 (rounded to the nearest integer).

More information will be provided in the syllabus.

Reworks

- After you receive HW feedback, you can rework the assignment during the 1-week "rework period" for that assignment to improve your grade (up to 1 point, so 2/3 may be updated to 3/3, but 1/3 can only be updated to 2/3)
- You start with 1 free reworks to use throughout the term
- For every 3 lectures you attend and fill out an exit form for, you can earn an extra rework
- Max of 2 reworks total per assignment

Collaboration Policy

We take collaboration and academic integrity very seriously, and have a thorough overview of expectations and rules around collaboration you are expected to be read and follow here.

In short, you are welcome to collaborate informally with students, but may **not look at each others code** and may **only write your own code** (Tuesday labs are an exception), as long as you follow the course policies outlined.

We take this no-collaboration policy very seriously for assignments and run submissions through plagiarism detection tools. Ask if you are unsure!

Teaching Challenges

Any CS1 course is hard to teach because of the enormous variation in the programming experience of students taking the class

- Probably greater than in any other subject

As an example, let’s look at hypothetical CS 1 student profiles...

Hypothetical Students in CS 1

Student #1: "I'm interested in learning how to program, but I've never done any programming of any kind before."

Student #2: "I did some programming in high school. It was fun, but I didn't feel like I really understood what I was doing. I'd like to learn to program the right way."

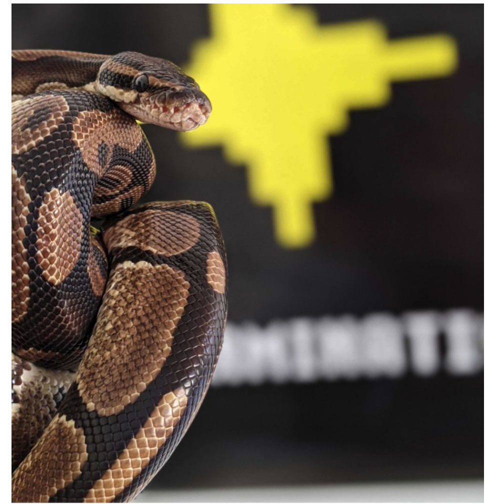
Student #3: "I've been programming since I was 5 years old. I wrote my own programming language when I was 11. When I was 16, I wrote my own operating system and sold it to Google for $50M. I'm at Caltech to give me something to do between IPOs."

This Course

This course is ideal for students #1 and #2.

Student #3 should probably take CS 2 (next term)

Introduction to Python



Learning Objectives

- Identify the difference between a code **editor** and **interactive Python shell**
- Know how to execute Python code on 1) a .py program file and 2) in the Python shell
- Learn about different **types** in Python and how to use them in **expressions**
- Understand how to store information using **variables**

What is Python?

- Programming language named after “Monty Python’s Flying Circus”
- Designed by Guido van Rossum starting in 1991
- Most recent version is Python 3.9.7
    - Make sure you have the current version installed! There are non-subtle differences from Python 2.X
    - Version 3.8+ is also fine
- Read all about the history here :)

Important Note: Python 2 vs. Python 3

There was a significant change in Python's syntax when Python version 3.0 was introduced

Almost everyone has made the switch by now, but you will still occasionally see Python 2 code in the wild

The Python interpreter program for Python 3 is called `python3` while the one for Python 2 is called `python2`

- Depending on the computer, python could refer to either one of them

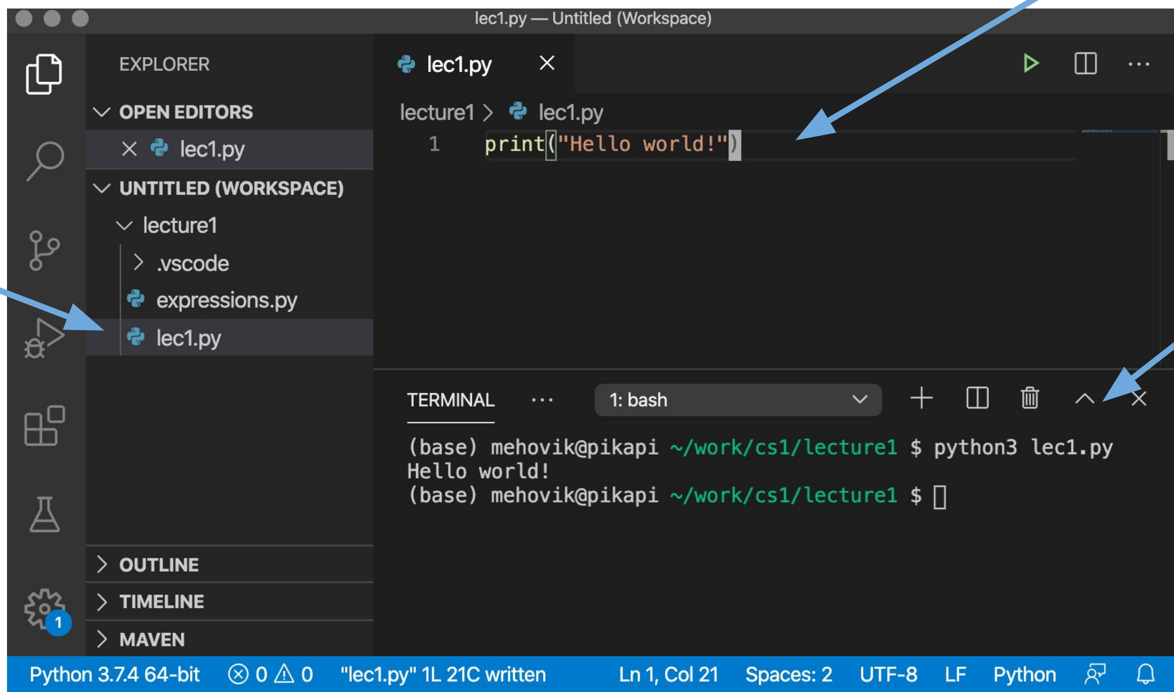
Always use `python3` instead, otherwise programs may not run correctly

Your First Python Program (in VSCode)

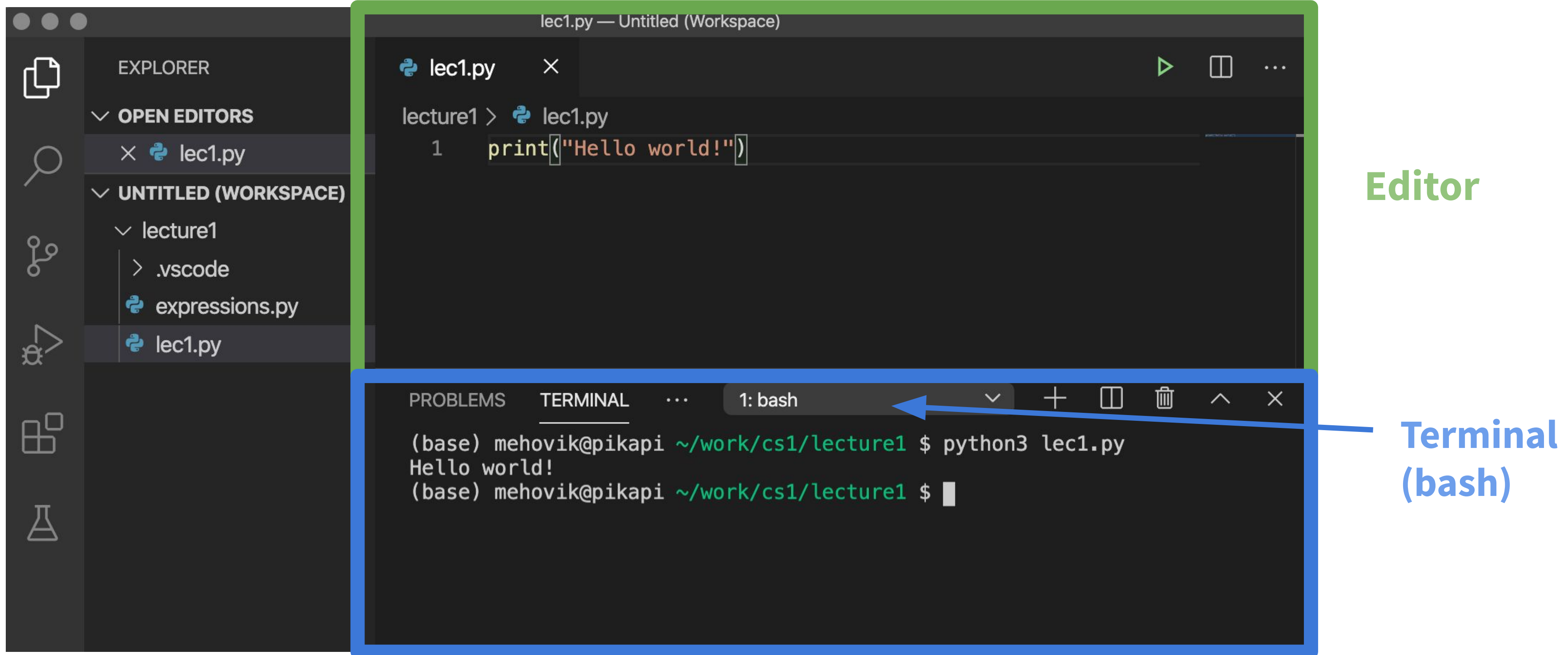
1. Write a program in a text editor (e.g. VSCode). This is called the **source code**.

2. Save the source code as a file ending in ".py" (such as `lec1.py`)

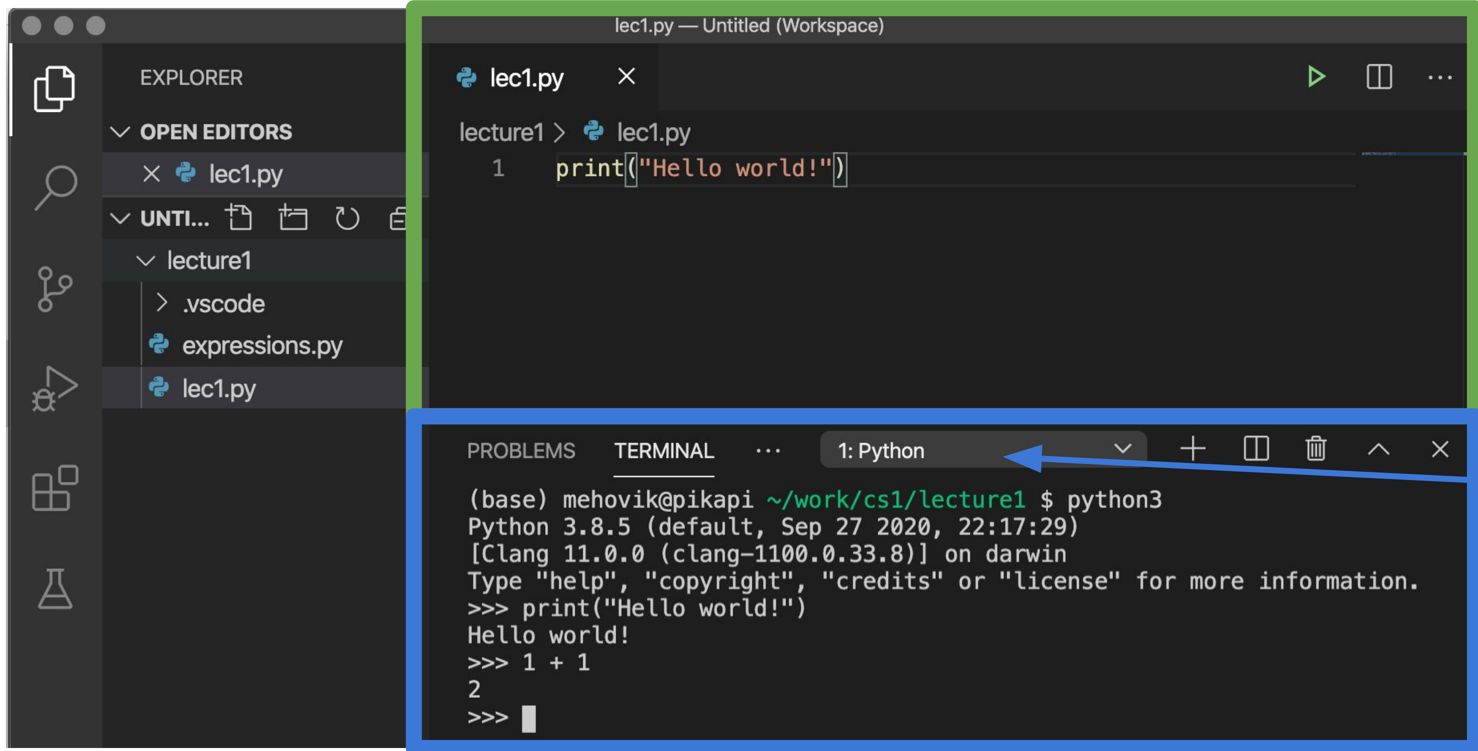
3. Execute the program by running the `python3` command in a **terminal**



Using the Terminal to Run a .py Program



Alternative: Invoking the Python Shell



Editor

Python shell (with >>>)

Writing vs. Running Code

The editor is where you write your code

The Python shell is where you can interactively experiment with Python code. This shell is entered from a terminal

Note: Using VSCode is not the only way to write programs in Python!

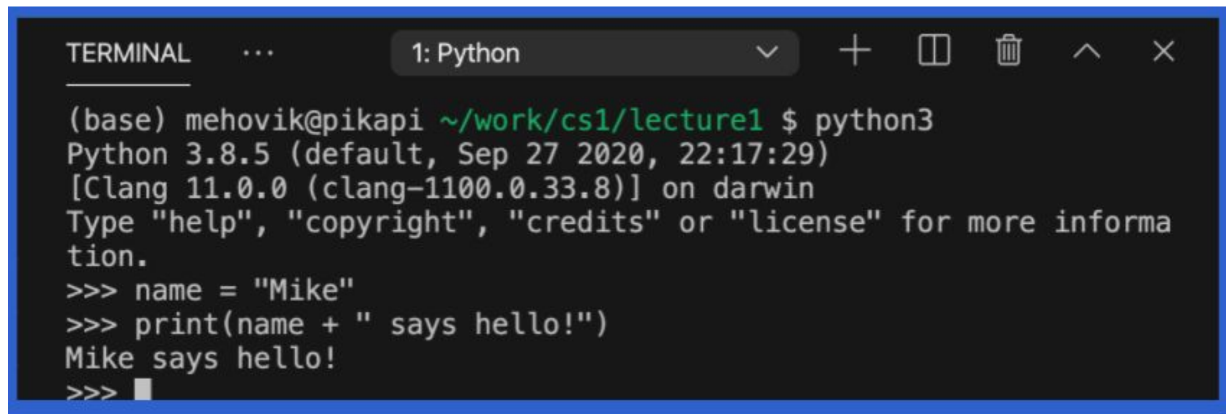
- VSCode has a convenient editor with a terminal with a Python shell, and doesn't have the feature overhead of other tools
- You can also use your computer's bash terminal if you would like

The Python Shell

The Python “shell” is just an interactive interpreter of Python code It prints a prompt (>>>) and waits for you to enter Python source code

Then it evaluates your code, prints the result, and prints another prompt, etc.

We will use this a lot in our examples



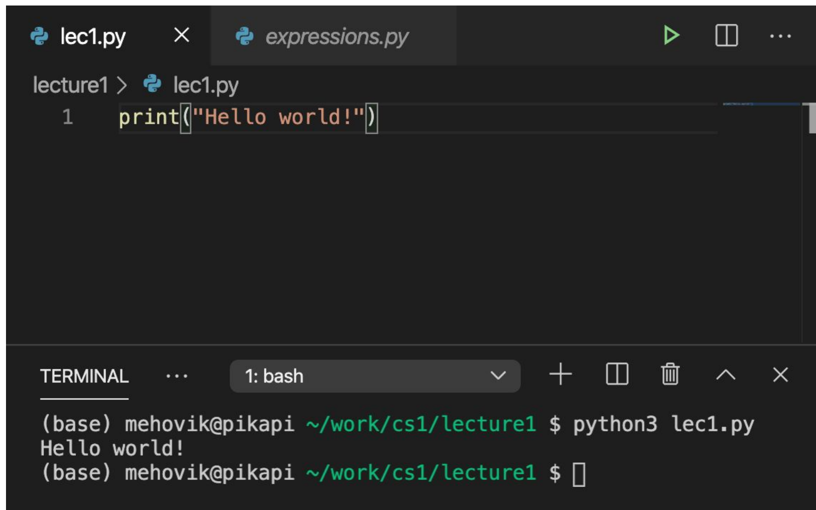
Python shell (with >>>)

Our First Line of Python

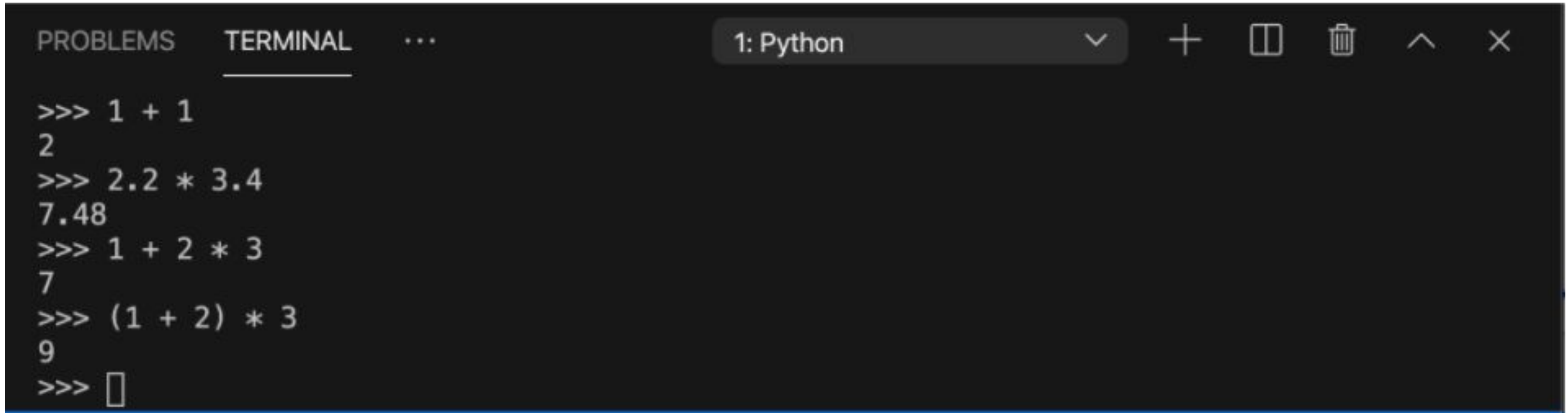
In Python, you can print output to the terminal using the **`print`** function

Traditionally, the first words you learn to write in any new programming language are "Hello world"

*Note: You don't need to use **`print`** when using the Python interpreter, as this **interprets** any code you enter immediately and outputs the result to the console*



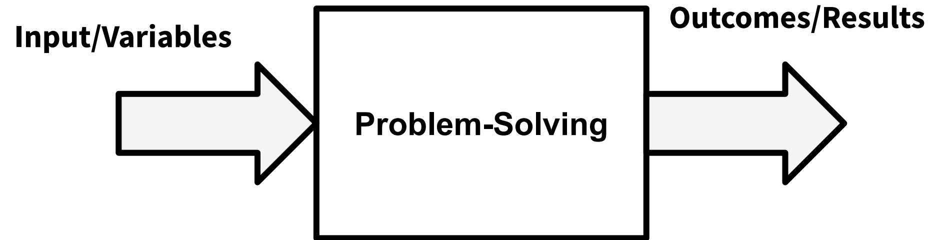
Replacing Your Calculator with Python



What is a Program, Really?

Even if you do not have any programming experience, you have all solved problems in the real-world:

- Math problems
- Chemistry/Biology/Physics problems
- Budgeting
- Planning your schedule for this term
- Applying to Caltech
- Prioritizing your commitments/obligations
- Making arguments/finding compromises with people
- Finding the best deal for a new computer
- Solving puzzles/strategizing in board or video games
- …



Fundamentals of Programming

Programming is all about formalizing problem-solving using a language of choice (we happen to use Python in CS 1)

This week, we'll introduce the fundamentals of programming to solve a variety of problems:

- Arithmetic and expressions
- Variables and assignments
- Datatypes
- Functions
- Scope
- Program Decomposition

Preview: Data Types

Data in programming languages is subdivided into different "types":

- integers: `0 -43 1001`
- floating-point numbers: `3.1415 2.718`
- boolean values: `True False`
- strings: `'foobar' 'hello, world!'`
- and many others

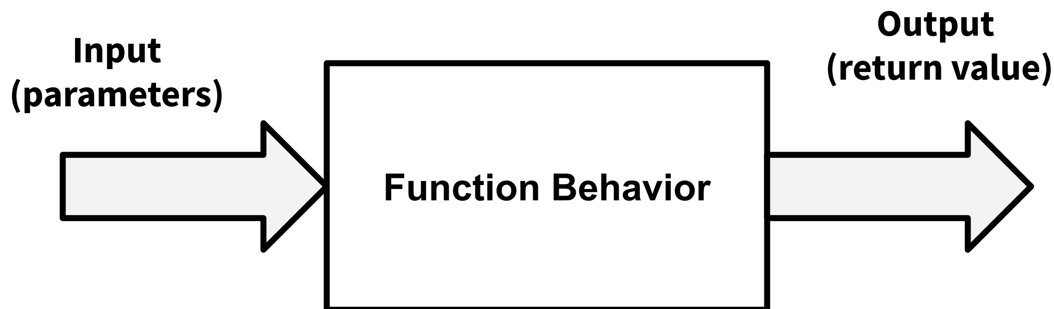
Knowing this, let's start with identifying "functions in the real world"...

Preview: Anatomy of a Function

A function is like a machine to perform tasks and possibly return some result

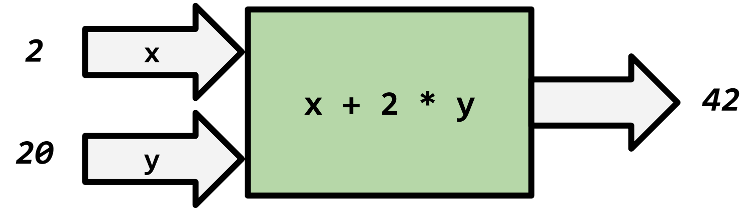
Every function has:

- Behavior (body)
- Parameters (optional)
- Return value (optional)

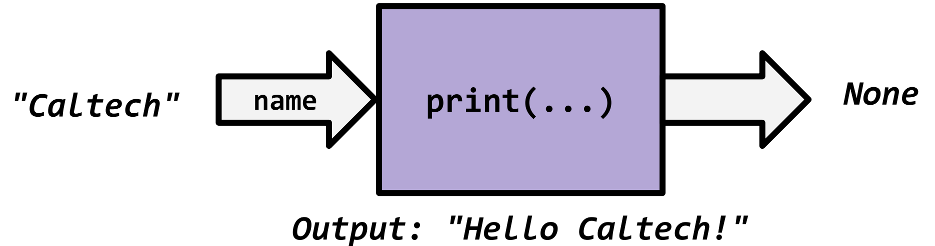


Preview: Functions as Machines

```
1 # Defining the function
2 def f(x, y):
3     return x + 2 * y
4
5 # Calling the function
6 ans = f(2, 20)
```




```
1 # Defining the function
2 def say_hello(name)
3   print("Hello", name, "!")
4
5 # Calling the function
6 say_hello("Caltech")
```



Activity

In Discord #lecture, share your response to the following question:

What is an example real-world problem you could model as a function of input to output?

Some examples:

- Given a temperature in Fahrenheit, convert to Celsius
- Given a unit in feet (ft), convert to meters (m)
- Given a birthday, determine the age in years
- Given a Pokemon type, determine its weakness
- Given a favorite music genre, provide 10 recommended Spotify songs
- …

Coming Attractions

On Wednesday, we will:

- Briefly go over arithmetic, data types, and variables/assignment
- Learn about some useful built-in Python **functions**
- Introduce “packaging” code into our own reusable functions
- Learn about variable **scoping** with functions

Action Items

- Make sure you have access to Canvas, Discord, and CodePost
- **Indicate your Tuesday Lab availability on Discord**
- Read the syllabus and collaboration policies
- Read this week's readings before Wednesday
- Complete the student information survey on Discord
- Enjoy your first week of Spring term!

Preview to Lecture 2

Arithmetic and Expressions

Arithmetic expressions contain numbers (operands) combined with symbols (operators) which compute values given the numbers

Operators: + - * / etc.

Numbers can be integers (no decimal point) or floating-point (with decimals)

- Floating-point is an approximation to real numbers

Operator Precedence

What does `1 + 2 * 3` mean?

It could mean

- `1 + (2 * 3)`
- `(1 + 2) * 3`

Computer languages have precedence rules to determine meaning of ambiguous cases

Operator Precedence

What does `1 + 2 * 3` mean?

It could mean

- `1 + (2 * 3)` Correct!
- `(1 + 2) * 3`

Computer languages have precedence rules to determine meaning of ambiguous cases

Here, * has higher precedence than +, so the first meaning is correct

Operator Precedence

In general, + and - have lower precedence than * and /

The ** (exponentiation) operator is even higher precedence than * and /

```
>>> 2 * 3 ** 4
162
```

Use parentheses to force a different order of evaluation if you need it

```
>>> (2 * 3) ** 4
1296
```

Data Types

Data in programming languages is subdivided into different "types":

- integers: `0 -43 1001`
- floating-point numbers: `3.1415 2.718`
- boolean values: `True False`
- strings: `'foobar' 'hello, world!'`
- and many others

Preview: Types

In Python, the same variable can hold data of different types at different times:

```
>>> a = 'foobar'
>>> a
'foobar'
>>> a = 3.1415926
>>> a
3.1415926
```

What might be an issue with this?

Variables and Assignment

Often, we want to give names to quantities

In Python, use the = (assignment) operator to do this:

```
>>> salary = 18.5
```

From here on, `salary` stands for `18.5`

```
>>> salary * 20

370
```

Variables and Assignment

Names assigned to can be reassigned:

```
>>> salary = 18.5
>>> salary
18.5
>>> salary = 30
>>> salary
30
```

Variables and Assignment

Names of variables ("identifiers") can only consist of the letters a-z, A-Z, the digits 0-9, and the underscore (_)

Identifiers also cannot start with a digit (avoids confusion with numbers)

Identifiers can't contain spaces!

Note: Case of letters is significant

- Foo is a different identifier than foo

```
a = 10

b1 = 20

this_is_a_name = 30

&*%$2foo? = 40 # not valid!
```

Variables and Assignment

Can have expressions on the right-hand side of assignment statements:

```
>>> salary = 18.5
>>> weekly_salary = salary * 20
>>> weekly_salary
370
```

The expression is terminated by the end of the line

Variables and Assignment

Can use results of previous assignments in subsequent ones:

```
>>> x = 15
>>> y = x * 5
>>> y
75
>>> z = x + y
>>> z
90
>>> z = z + 10
>>> z
100
```

Variables and Assignment

Evaluation rule for assignment statements:

1. Evaluate the right-hand side
2. Assign the resulting value to the variable on the left-hand side

This explains why `z = z + 10` works:

- previously, z was 90
- evaluate `z + 10` to 100
- assign 100 to z (new value)

Variables can vary!

Types

Data in programming languages is subdivided into different "types":

- integers: `0 -43 1001`
- floating-point numbers: `3.1415 2.718`
- boolean values: `True False`
- strings: `'foobar' 'hello, world!'`
- and many others

Types

In Python, the same variable can hold data of different types at different times:

```
>>> a = 'foobar'
>>> a
'foobar'
>>> a = 3.1415926
>>> a
3.1415926
```

What might be an issue with this?

Functions

A function takes some input data and transforms it into output data

Functions must be defined and then called with the appropriate arguments

A few functions are built-in to Python so we don’t have to define them ourselves

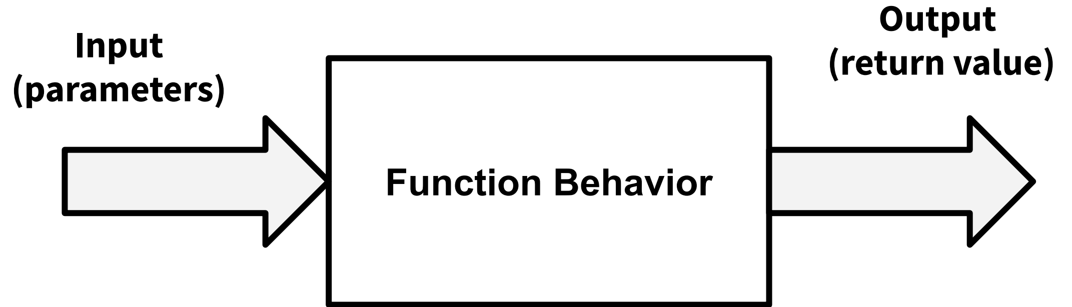
- `print(x)`
- `input(x)`
- `type(x)`
- `int(x), float(x), str(x)`
- `help()`

Anatomy of a Function

A function is like a machine to perform tasks and possibly return some result

Every function has:

- Behavior (body)
- Parameters (optional)
- Return value (optional)



Defining and Calling Functions

Functions may have parameters passed to help generalize functionality and may also specify a return value with the return keyword (**None** if no return specified)

Definition Syntax:

```
def name(<parameters>):
    <body>
    return <value> # optional
```

Definition Examples:

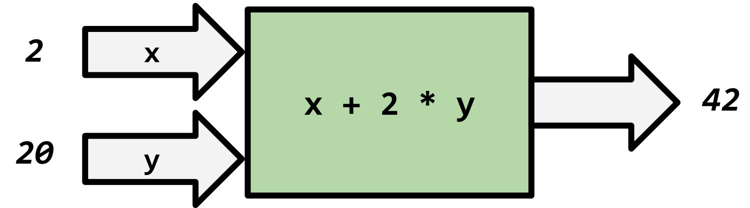
```
def say_hello(name):
    print("Hello", name, "!")

def f(x, y)
    return x + 2 * y
```

Function Call Examples:

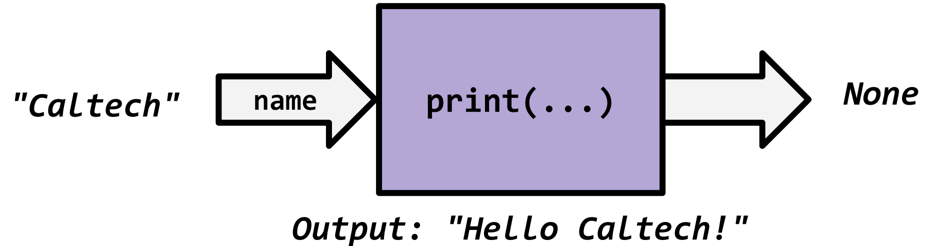
```
say_hello("world")   # Hello world!
say_hello("Caltech") # Hello Caltech!
ans = f(2, 20)       # ans == 42
```

Functions as Machines

```python
1 # Defining the function
2 def f(x, y):
3     return x + 2 * y
4
5 # Calling the function
6 ans = f(2, 20)
```




```python
1 # Defining the function
2 def say_hello(name)
3    print("Hello", name, "!")
4
5 # Calling the function
6 say_hello("Caltech")
```



Scope is Important!

So far, our variables have been defined top-down - later assignments shadow will shadow earlier ones.

Functions introduce their own **local** scope - **variables inside functions only exist in during the lifetime of a function call**.

```
1  def f(x, y)
2      return x + 2 * y
3
4  ans1 = f(2, 20) # 42
5  ans2 = f(x, 20) # error! x is not in scope here
```

Parameters vs. Arguments

Formal parameters are simply names for the argument values passed in a function call. The **position of arguments** will determine what formal parameter name they are assigned.

They have no relationship to other variable names in the program and will override other variables if there is a naming conflict.

```
1  def f(x, y)
2      return x + 2 * y
3
4  a = 2
5  b = 20
6  ans1 = f(a, b)
7  ans2 = f(b, a) # b and a are mapped to x and y in f, respectively
```

Practice

What is the result of executing the following program? (PythonTutor demo)

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print("x: ", x, "y: ", y, "z: ", z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

Practice

What is the result of executing the following program? (PythonTutor demo)

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print("x: ", x, "y: ", y, "z: ", z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

```
Output:
x: 1 y: 2 z: 3
x: 3 y: 1 z: 4
```