Bonus Slide

The First Program

Babbage hired Ada Lovelace to translate lecture notes on the Analytical Engine in 1843 (Lecture 1-1).

She added extensive notes to her translation with her own thoughts, including an example program that could calculate Bernoulli numbers. This was the first program!

She also posited that the Analytical engine could work with 'things beside number', such as music. She was the first person to recognize the potential for computation to be used outside of mathematics.

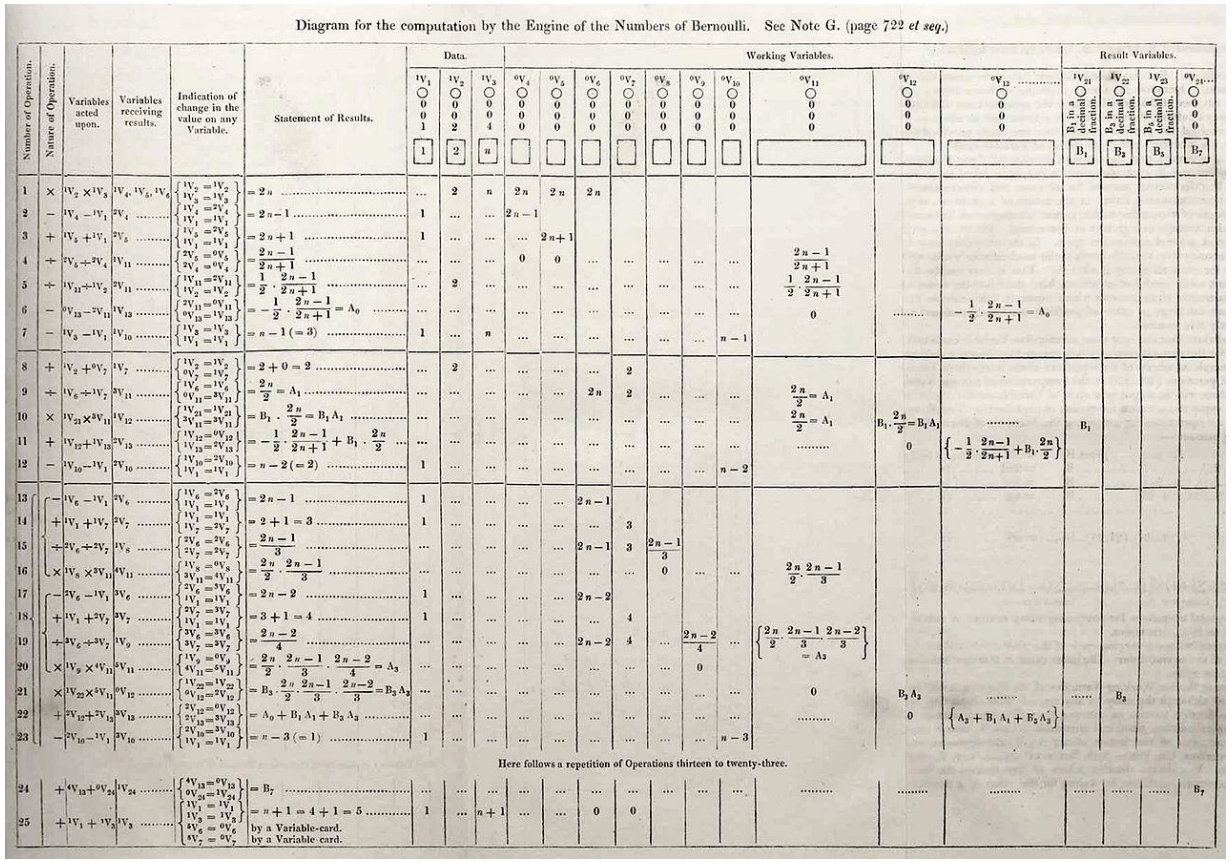


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data. $^1V_1$ 1	Data. $^1V_2$ 2	Data. $^1V_3$ n	Working Variables. $^0V_4$	$^0V_5$	$^0V_6$	$^0V_7$	$^0V_8$	$^0V_9$	$^0V_{10}$	$^0V_{11}$	$^0V_{12}$	$^0V_{13}$	Result Variables. $^1V_{21}$ B_1	$^1V_{22}$ B_3	$^1V_{23}$ B_5	$^0V_{24}$... B_7
1	×	$^1V_2 \times {}^1V_3$	$^2V_4$, $^1V_5$, $^1V_6$	$\begin{cases} {}^1V_2 = {}^1V_2 \\ {}^1V_3 = {}^1V_3 \end{cases}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^2V_4 - {}^1V_1$	$^2V_4$	$\begin{cases} {}^1V_4 = {}^2V_4 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-1$	1	...	...	$2n-1$													
3	+	$^1V_5 + {}^1V_1$	$^2V_5$	$\begin{cases} {}^1V_5 = {}^2V_5 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n+1$	1	...	...	...	$2n+1$												
4	+	$^2V_5 \div {}^2V_4$	$^1V_{11}$	$\begin{cases} {}^2V_5 = {}^0V_5 \\ {}^2V_4 = {}^0V_4 \end{cases}$	$= \frac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\frac{2n-1}{2n+1}$						
5	+	$^1V_{11} \div {}^1V_2$	$^2V_{11}$	$\begin{cases} {}^1V_{11} = {}^2V_{11} \\ {}^1V_2 = {}^1V_2 \end{cases}$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\frac{1}{2} \cdot \frac{2n-1}{2n+1}$						
6	−	$^0V_{13} - {}^2V_{11}$	$^1V_{13}$	$\begin{cases} {}^2V_{11} = {}^0V_{11} \\ {}^0V_{13} = {}^1V_{13} \end{cases}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	.....	$-\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$				
7	−	$^1V_3 - {}^1V_1$	$^2V_{10}$	$\begin{cases} {}^1V_3 = {}^1V_3 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-1\ (=3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^1V_2 + {}^0V_7$	$^1V_7$	$\begin{cases} {}^1V_2 = {}^1V_2 \\ {}^0V_7 = {}^1V_7 \end{cases}$	$= 2+0=2$	...	2	...	...	...	...	2	...	...	...							
9	+	$^1V_6 \div {}^1V_7$	$^2V_{11}$	$\begin{cases} {}^1V_6 = {}^1V_6 \\ {}^0V_{11} = {}^2V_{11} \end{cases}$	$= \frac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\frac{2n}{2} = A_1$						
10	×	$^1V_{21} \times {}^2V_{11}$	$^1V_{12}$	$\begin{cases} {}^1V_{21} = {}^1V_{21} \\ {}^2V_{11} = {}^2V_{11} \end{cases}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\frac{2n}{2} = A_1$	$B_1 \cdot \frac{2n}{2} = B_1 A_1$	.....	B_1			
11	+	$^1V_{12} + {}^1V_{13}$	$^2V_{13}$	$\begin{cases} {}^1V_{12} = {}^0V_{12} \\ {}^1V_{13} = {}^2V_{13} \end{cases}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{-\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \frac{2n}{2}\right\}$				
12	−	$^1V_{10} - {}^1V_1$	$^2V_{10}$	$\begin{cases} {}^1V_{10} = {}^2V_{10} \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-2\ (=2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13		$^1V_6 - {}^1V_1$	$^2V_6$	$\begin{cases} {}^1V_6 = {}^2V_6 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-1$	1	...	...	...	...	$2n-1$											
14	+	$^1V_1 + {}^1V_2$	$^2V_7$	$\begin{cases} {}^1V_1 = {}^2V_1 \\ {}^1V_7 = {}^2V_7 \end{cases}$	$= 2+1=3$	1	...	...	...	...	...	3										
15	÷	$^2V_6 \div {}^2V_7$	$^1V_8$	$\begin{cases} {}^2V_6 = {}^2V_6 \\ {}^2V_7 = {}^2V_7 \end{cases}$	$= \frac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\frac{2n-1}{3}$									
16	×	$^1V_8 \times {}^2V_{11}$	$^4V_{11}$	$\begin{cases} {}^1V_8 = {}^0V_8 \\ {}^2V_{11} = {}^4V_{11} \end{cases}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\frac{2n}{2} \cdot \frac{2n-1}{3}$						
17		$^2V_6 - {}^1V_1$	$^3V_6$	$\begin{cases} {}^2V_6 = {}^3V_6 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-2$	1	...	...	...	...	$2n-2$											
18	+	$^1V_1 + {}^2V_7$	$^3V_7$	$\begin{cases} {}^2V_7 = {}^3V_7 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 3+1=4$	1	...	...	...	...	...	4										
19	÷	$^3V_6 \div {}^2V_7$	$^1V_9$	$\begin{cases} {}^3V_6 = {}^3V_6 \\ {}^2V_7 = {}^2V_7 \end{cases}$	$= \frac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\frac{2n-2}{4}$	...	$\left\{\frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3}\right\}$ $= A_2$							
20	×	$^1V_9 \times {}^4V_{11}$	$^5V_{11}$	$\begin{cases} {}^1V_9 = {}^0V_9 \\ {}^4V_{11} = {}^5V_{11} \end{cases}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = A_3$	...	...	...	...	...	...	...	...	...	0							
21	×	$^1V_{22} \times {}^5V_{11}$	$^6V_{12}$	$\begin{cases} {}^1V_{22} = {}^1V_{22} \\ {}^0V_{12} = {}^2V_{12} \end{cases}$	$= B_3 \cdot \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	.....	.....	B_3		
22	+	$^2V_{12} + {}^2V_{13}$	$^3V_{13}$	$\begin{cases} {}^2V_{12} = {}^0V_{12} \\ {}^2V_{13} = {}^3V_{13} \end{cases}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	.....	0	$\{A_0 + B_1 A_1 + B_3 A_3\}$				
23	−	$^2V_{10} - {}^1V_1$	$^3V_{10}$	$\begin{cases} {}^2V_{10} = {}^3V_{10} \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-3\ (=1)$	1	...	...	...	...	...	...	...	...	$n-3$							
Here follows a repetition of Operations thirteen to twenty-three.																						
24		$^{4V_{13}} + {}^{0V_{24}}$	$^1V_{24}$	$\begin{cases} {}^4V_{13} = {}^0V_{13} \\ {}^0V_{24} = {}^1V_{24} \end{cases}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	.....	.....	.....	.....	.....	.....	B_7
25		$^1V_1 + {}^1V_2$	$^1V_3$	$\begin{cases} {}^1V_1 = {}^1V_1 \\ {}^1V_3 = {}^1V_3 \\ {}^1V_6 = {}^6V_6 \\ {}^1V_7 = {}^6V_7 \end{cases}$	$= n+1 = 4+1 = 5$ ..... by a Variable-card. by a Variable-card.	1	...	$n+1$	...	...	0	0	...	...	...	.....	.....	.....	.....	.....	.....	B_7

Function Calls

15-110 – Monday 08/31

Announcements

- Hw1 was due **today at noon**

- Hw2 has been released

Learning Objectives

- Use **function calls** to run pre-built algorithms on specific inputs
- Identify the **argument(s)** and **returned value** of a function call
- Use **libraries** to import functions in categories like math and randomness
- Use **testing** to verify whether a Python program produces correct results.

Repeating Actions is Messy

Sometimes we want to perform the same algorithm many times on different inputs.

For example, say we want to personalize a young child's reading material so that it uses their pet's name.

We could copy and paste the first bit of code, then change the necessary parts. But if we're sloppy this might cause errors.

```
pet1 = "Spot"
pet2 = "Stella"
pet3 = "Willow"

print("See " + pet1 + ". See " + pet1 +
      " run. Run, " + pet1 + ", run!")

print("See " + pet2 + ". See " + pet2 +
      " run. Run, " + pet2 + ", run!")

print("See " + pet3 + ". See " + pet1 +
      " run. Run, " + pet3 + ", run!")
```

Functions Represent Abstract Actions

A better approach is to put the core action being repeated into a **function**.

A function is a code construct that represents an algorithm. We can **define** a function once, then **call** it many times.

We can also use functions that have already been defined by Python.

Function Calls

Call Functions with Parentheses

We've already seen how to call a function on a specific input, because `print` is just a function! This is done using **parentheses**.

```
functionName(input1, input2, ...)
```

The number of inputs provided inside the parentheses depends on how many inputs the function expects. Each input should be an **expression**.

A Few New Functions

To help us explore how functions work, let's introduce a few new functions. These are **built-in functions**, like `print`; that means we can call them in Python directly.

```
abs(-2) # 2; absolute value
pow(2, 3) # 8; raises a number to the given power
round(12.4567, 2) # 12.46; rounds to the given # sig digs
```

A Special Function

There's another built-in function that works differently from the others. `input(msg)` displays a message in the interpreter, lets the user type a response in the interpreter, then stores the response as a string when the user presses enter.

```
input("Enter your name: ") # evaluates to whatever the user types
```

This will make it possible to write **interactive** programs more easily! This will also let the user **enter data** interactively.

Type Functions

There are a few other built-in functions that are helpful to know, as they let you change the type of data values. This is called **type-casting**, and it is especially useful when you need to change the type of user input.

```
int("4") # 4; converts a value to an integer
float(3) # 3.0; converts a value to a float
str(98.9) # "98.9"; converts a value to a string
bool(0) # False; converts a value to a Boolean

type(4 + 3.0) # float; returns the type of the eventual value
# uses the names we covered before – int, float, str, bool
```

Components of Functions

The functions we call may have two core components:

Argument(s) – the values that are provided inside the parentheses, the **input**

Returned Value – what the function evaluates to after running, the **output**

Arguments Provide the Input

The specific inputs we provide to a function are called **arguments**. These are like the specific bread, peanut butter, and jelly we used in the PB&J algorithm. In the function call `abs(4)`, the argument is `4`.

Arguments are separated by commas and placed between the parentheses of the function call. Functions can require as many (or as few) arguments as needed.

The **positions** of the arguments usually have meaning. In `pow(2, 3)`, the first argument is the base and the second argument is the exponent. In other words, `pow(2, 3)` and `pow(3, 2)` mean two different things.

Receive Output as Returned Value

When a built-in function takes its arguments and runs through its algorithm, we cannot see what it is doing.

When the function is done, it sends back an output as a **returned value**. We usually say a function **returns** a value. This value substitutes in for the function call the same way a variable's value substitutes in for the variable.

For example, the returned value of `pow(2, 3)` is `8`.

Function Calls Follow Order of Operations

Function calls evaluate to a single returned value; that means they are **expressions**. Therefore, we can **nest** function calls inside other expressions the same way we nest basic values and operations.

```
round(pow(abs(-12), 1/2), 2)
```

Just like in math, functions follow order of operations using parentheses. Start by evaluating the inner-most expressions, `abs(-12)` and `1/2`. Then evaluate the call to `pow`; finally, evaluate the call to `round`.

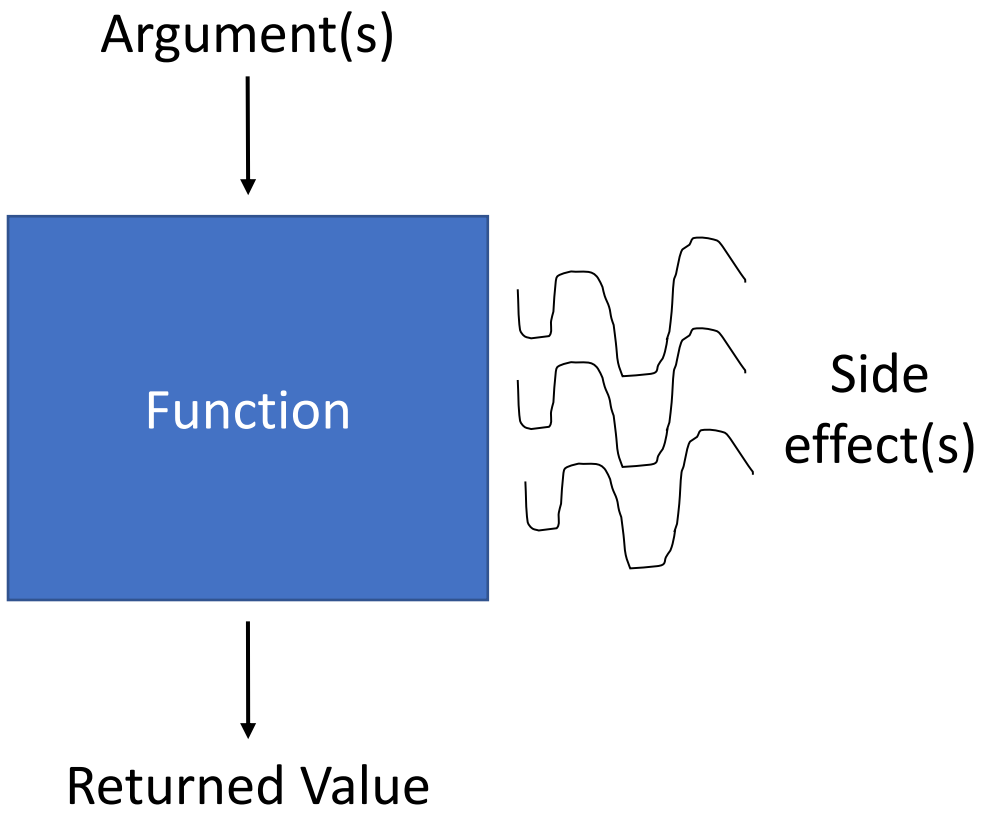
Side Effects Show Change

Recall that a program has a **state** that holds the current information that the program knows (what has been printed, what values do variables hold).

Function calls themselves are expressions, as they evaluate to a data value (the returned value). But sometimes a function changes the program state in an observable way as it is running; for example, it might display values in the interpreter, or modify a file, or produce graphics. This is called a **side effect**.

If we call `pow(2, 3)`, there is no observable side effect. However, `input("How are you?")` has an observable side effect: it prints a message to the screen and pauses the program until the user responds. `input` also has a returned value – the message typed by the user.

Function Call Process



`print` works differently

Let's take a moment to talk about a function that works in a particularly confusing way: `print`.

`print` takes arguments (the values between the parentheses). It produces a side effect when it displays the result of concatenating those values to the interpreter.

What is `print`'s returned value? It could be the displayed value, but that would let us do weird things like:

```
x = print(2) + 2 # sets x = 4 # but not really!
```

We probably don't want that. Instead, we'll say that `print` has **no explicit output**. But it's not that simple!

Missing Returned Values are None

If a function produces no explicit output, it still has a returned value – we need *something* to store in a variable or display. That value is the built-in value `None`.

`None` means that there was no explicit output to be returned. Like `True` and `False`, its meaning is built into Python, so it does not need quotes.

If you try to set a variable to the returned value of a `print` call, you'll find that the variable holds `None`; print **always** returns `None`. Note that `None` does not show up in the interpreter unless you explicitly `print` it; the interpreter just shows a blank instead.

```
>>> None
>>> print(None)
 None
>>>
```

Activity – Identify the Function Call Parts

Consider the following two function calls. For each function call, what are its **argument(s)** and **returned value?** Does it have any observable **side effect(s)**?

```
round(3.1415, 1)
```

```
print("15" + "-" + "110")
```

Libraries

Import Adds Code from Libraries

The Python language has a ton of pre-built functions, but most aren't included in the built-in package (the one available by default). Most of the functions are organized into separate **libraries**.

To use a function from a library, you must **import** the library. This makes it possible to access the functions and variables in that collection. You can do this with the code:

```
import libraryName
```

Library Documentation Organizes Functions

How can you determine which functions exist in which libraries? Read the **documentation**!

All the Python libraries have documentation online that describes which functions are available and what they do. Find it by going to docs.python.org/3/ .

There are a great many libraries and functions, so it's better to check the documentation as needed than to try to memorize all the functions that exist.

Importing the math Library

For example, we can import the **math** library to add more mathematical capabilities. Note that we must put `math.` in front of each function or variable name we use, to specify it came from that library.

```
import math
math.ceil(6.5) # 7; ceiling of a float number
math.log(64, 2) # 6.0; finds the log of 64 with base 2
math.radians(90) # 1.570…; converts degrees to radians
math.pi # 3.141…; it's π!
```

Importing the random library

Importing libraries lets us get more creative with programming. For example, the **random** library lets us generate random numbers, which can help produce novel behavior.

```
import random
random.randint(1, 10) # picks a random int between 1-10 inclusive
random.random() # picks a random float between 0-1
```

Testing is Believing

Why do we need testing?

Code from any source, a classmate, a teammate, or GenAI, needs to be verified before you trust it.

Humans make mistakes: there's often a gap between intent (*what we meant to write*) and implementation (*what we actually wrote*).

GenAI is probabilistic, not deterministic. It can confidently produce code that looks right but isn't.

In industry, testing is a serious discipline (often entire job roles) because bugs have real consequences (as we will see throughout the semester): lost revenue, security breaches, even physical safety incidents.

Our Motto:

Code is wrong
until proven otherwise!

How do we test?

Testing means giving your code different inputs and checking whether the output matches what you expect.

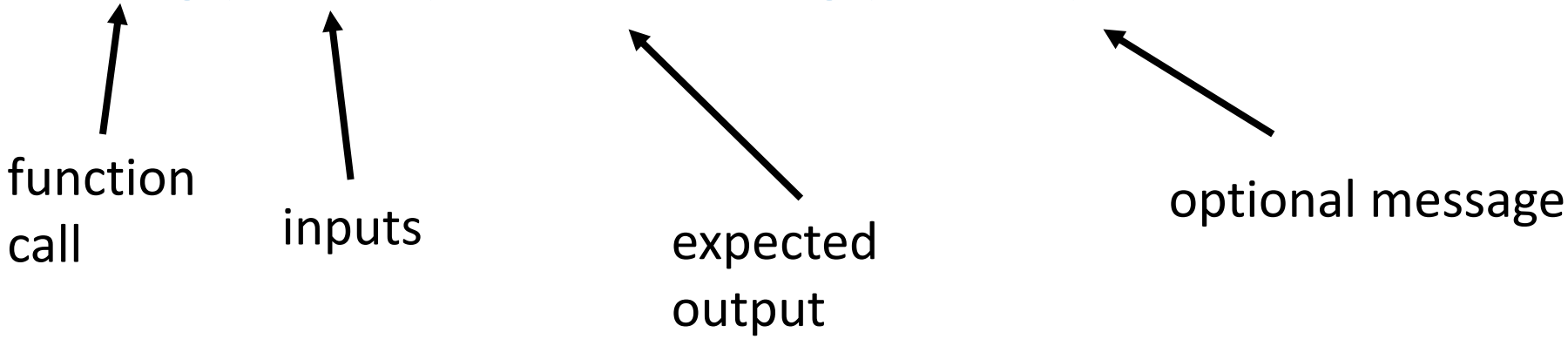
To test our functions, we will:

- Decide on a set of inputs to try
- Predict the expected output for each input
- Call the function with the inputs and compare its actual output to what we expected

`assert` Statements Check Correctness

An `assert` statement takes a Boolean expression. If the expression evaluates to `True`, the statement does nothing. If it evaluates to `False`, the program crashes with an `AssertionError`.

```
assert avg(20, 4) == 5, "avg(20, 4) should be 5"
```



Case Study

Prof. Franceska's close personal friend Megan Thee Stallion has given her a file called **megtheestallion.py** with a **count_a** function that says:

```
"""
Takes a string as input and counts how many lowercase 'a' characters
this string contains. Note that uppercase 'A' does not count.
    Example: count_a("banana") returns 3
             -> because lowercase 'a' appears 3 times
"""
```

Let’s think about how we would test this function.

- What inputs would you call the function with?
- What outputs would you expect the function to give you?

Let’s apply it in practice:

Let’s download both files. We will work with the file `verify.py`, where we'll import `megtheestallion.py` . Let’s start testing!!

First, call the function in the shell. Does it work as you expected?

Next, use an `assert` statement to check for correctness:

```
assert megtheestallion.count_a("banana") == 3, "Expected 3 a's in 'banana'"
```

Learning Objectives

- Use **function calls** to run pre-built algorithms on specific inputs
- Identify the **argument(s)** and **returned value** of a function call
- Use **libraries** to import functions in categories like math and randomness
- Use **testing** to verify whether a Python program produces correct results.