Lecture 04

Further Python

COSC110

Introduction to Programming and the UNIX environment

Requirements for Programming

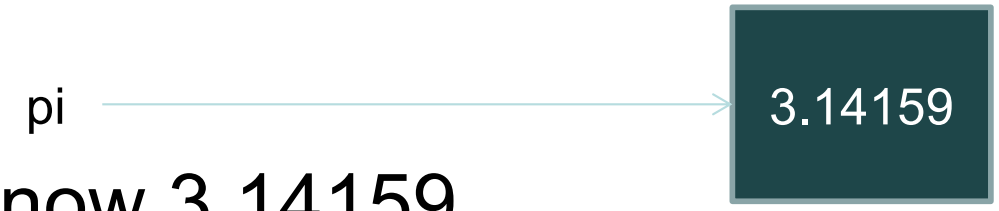
- ~~Input~~
- ~~Output~~
- ~~Mathematics~~
- ~~Conditional execution~~
- Repetition

Outline

- Iteration
  - Reassignment
  - The while statement
  - break

Reassignment

- Each variable is like a box in memory
  - `pi = 3.14159`



  - *pi* is now 3.14159
- If we reassign a value, the value in the box is replaced

  `pi = 3`

  - *pi* is now exactly 3

=

- = is used for assignment
  - `a = b` does not mean *a* and *b* are equal
    - It means *a* is set to the current value of *b*
  - Remember, statements like `7 = a` are not legal in Python
  - Even if variables are equal, that can change

```
a = 12
b = a # a and b are equal at the moment
a = 9 # a and b are no longer equal
```

Updating Variables

- We can reassign a variable such that its new value depends on the old value:

```
x = x + 1
```

```
x += 1
```

  - Get the current value of *x*, add 1, and then replace *x* with the updated value
  - If *x* is not already defined, we get a *NameError*
    - Need to initialise *x* first
      - E.g.,

```
x = 0
x = x + 1
```

- Updating a variable by adding 1 is an *increment*
- Updating a variable by subtracting 1 is a *decrement*

Iteration

- We often need to repeat identical (or similar) tasks
  - This is something computers do well
- Python has a construct very similar to the *while* statement we saw with bash

while

- In Python, the *while* statement looks like:

```
n = 10
while n > 0:
    print(n)
    n -= 1
print("Countdown complete")
```

while structure

- Every *while* loop has the following format:

```
<initialisation>
while <condition>:
    <body>
<after loop>
```

- If the condition is *True*, the body of the loop is executed and we loop back to check the condition again

Infinite Loops

- Consider the following program:

```
n = 10
while n > 0:
    print(n)
print('Countdown complete')
```

- We need to ensure the loop will eventually change the condition to be false, or the loop will keep repeating forever

Collatz Conjecture

- It is not always easy to determine if a loop will eventually stop:

```
start = input('Enter a value: ')
n = int(start)
while n != 1:
    if n % 2 == 0: # n even
        n = n // 2
    else:          # n odd
        n = 3 * n + 1
```

break

- Sometimes you are partway through a loop before you realise it is time to leave the loop

```
while True:
    line = input('Exit? ')
    if line == 'Y' or line == 'y':
        break
    print("We'll loop again")
```

Example: Square Roots

- Newton's method for calculating square roots starts with an estimate, then iteratively improves the answer
- To find an approximation of the square root of *a*, given the initial estimate *x*, you can typically calculate a better estimate with:

```
y = (x + a/x) / 2
```

- If we repeat this process until the new estimate equals the old estimate, we know we've found an actual square root

Example: Square Roots

```
square = input("Enter value to find root: ")
a = float(square)
estimate = input("Enter initial estimate: ")
x = float(estimate)
while True:
  print(x)
  y = (x + a/x) / 2
  if y == x:
    break
  x = y
```

This code has a problem!

Representing Floats

- Computers use a number of bytes to represent a float
  - Not all numbers can be represented exactly
- Similar to us expressing numbers as decimals
  - E.g. represent 1/3 using a finite number of decimal places
- Because computers work in binary, there are numbers we can represent exactly in decimal that the computer can't represent as a float
  - You should never* compare floats for equality
    - Instead, check they are "close enough" to each other

* There are some instances where it is ok – but avoid unless you're sure

Example: Square Roots

```
epsilon = 0.001
square = input('Enter value to find root: ')
a = float(square)
estimate = input('Enter initial estimate: ')
x = float(estimate)
while True:
  print(x)
  y = (x + a/x) / 2
  if abs(y-x) < epsilon:
    break
  x = y
```

Repeating a Fixed Number of Times

- while loops repeat until the condition is *False*
- Sometimes you want to repeat a task a certain number of times
- You could use a construct similar to our *countdown* example
  - Instead of just counting down, it could perform the task before the decrement
- Python has a construct to make this type of loop easier
  - The *for* loop

for loops

```
for x in range(5):
  print('x = ', x)

for x in range(0, 5):
  print('x = ', x)

for x in range(3, 6):
  print('x = ', x)

for x in range(3, 9, 2):
  print('x = ', x)
```

Summary

- Reassignment allows us to update a variable's value
- while loops repeat until a condition is no longer true
- for loops repeat a certain number of times
- We should never compare float values for equality

Outline

- Lists
  - Lists as sequences
  - Traversing a list
  - List operations
  - Aliasing

Sequences

- It is often useful to have a sequence of values
  - For example, we might want a sequence that contains the name of each day of the week
- In Python, sequences can be created in a built-in data type called a list

```
['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday',
'Sunday']
```

Types in Lists

- An empty list

```
[]
```

- A list of integers:

```
[123, 543, 74, 432, 568]
```

- A list of floats:

```
[1.12, 54.3, 65.2, 23.4562]
```

- A list of strings:

```
['purple', 'monkey', 'dishwasher']
```

- A list of lists:

```
[[2, 7, 4], [1.1, 9.42], ['a', 'b']]
```

- A list with different types:

```
[123, 1.12, 'purple', [2, 7, 4]]
```

List Variables

- Lists can be assigned to variables:

```
> days = ['Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
> type(days)
  <class 'list'>
> print(days)
    ['Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
```

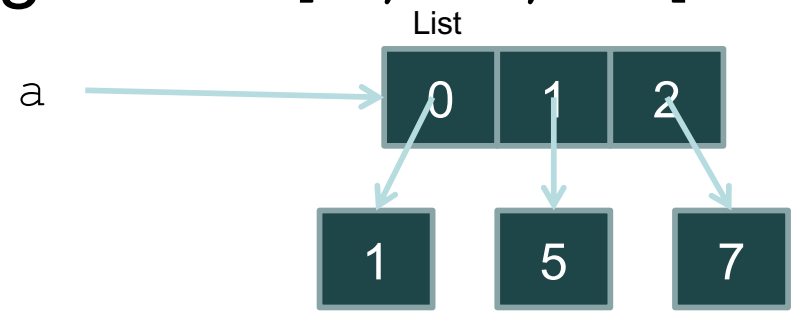
Accessing List Elements

- You can access list elements using the bracket operator
- E.g.

```
> days = ['Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
> days[0]
  'Monday'
> days[3]
  'Thursday'
```

List Indexes Start at 0

- Note that the first element of the list is the element in position 0
  - A list of length *n* has elements 0 to *n*-1
- This is because of how lists are stored in memory
  - E.g. `a = [1, 5, 7]`



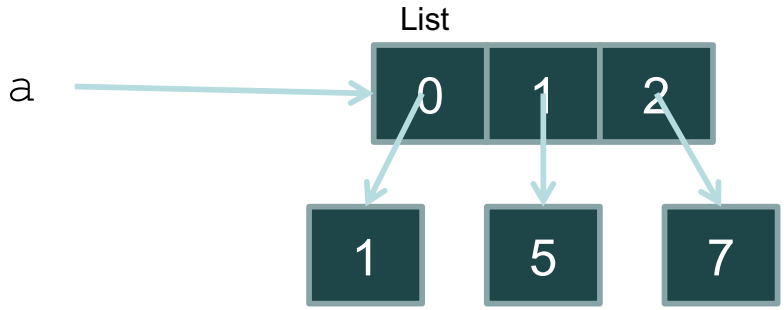
Changing List Elements

- You can also change list elements
- E.g.

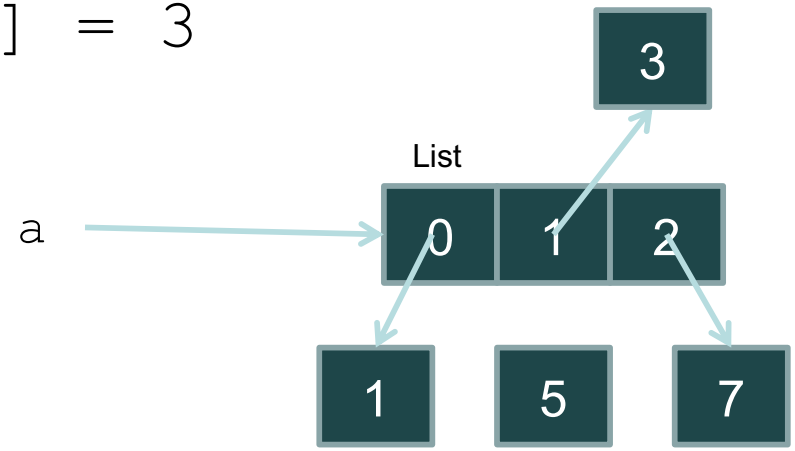
```
> days = ['Monday', 'Tuesday',
'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
> days[0] = 'Flurmsday'
> days[0]
  'Flurmsday'
```

Changing A List Element

- E.g. `a = [1, 5, 7]`



`a[1] = 3`



List Indexes

- Any integer expression can be used as an index
  - E.g., `a[0], a[5-2], a[a[0]]`
- If you try and read an element before the beginning of the list or after the end of a list, you get an *IndexError*
  - E.g., `a[1000000]`
- If the index is negative, it counts back from the end of the list
  - E.g., `a[-1]` is the last element of the list, `a[-2]` the second last, etc.

in Operator

- You can check to see if a particular value is in a list using the *in* operator

```
> weekend = ['Saturday', 'Sunday']
> 'Monday' in weekend
  False
> 'Sunday' in weekend
  True
```

Traversing a List

- We often want to go through each element in a list and perform some actions
- The most common way is using a *for* loop
  - E.g.

```
for day in days:
    print(day)
```

- *day* is updated each iteration through the loop to be the next element
- Looping over an empty list results in the loop's body never running

Updating List Elements in a Loop

- We can combine the built-in functions *range* and *len* to allow us to update/alter list elements as we loop:
- E.g

```
numbers = [2, 4, 6, 8]
for i in range(len(numbers)):
    numbers[i] = numbers[i] // 2
print(numbers)
```

- Gives output:

```
[1, 2, 3, 4]
```

Looping with Nested Lists

```
test = [1, [2, 3], 4]
for i in test:
  print(i)
```

Gives output:

```
1
[2, 3]
4
```

List Operations

- Similarly to strings, lists also have + and * operators
- + concatenates lists

```
> a = [1, 2, 3]
> b = [4, 5, 6]
> c = a + b
> print(c)
[1, 2, 3, 4, 5, 6]
```

- * repeats a list

```
> d = c * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

List Slices

- You can extract parts of a list using list slices

```
> numbers = [2, 4, 6, 8, 10]
> print(numbers[1:3])
[4, 6]
> print(numbers[:3])
[2, 4, 6]
> print(numbers[1:])
[4, 6, 8, 10]
```

Copying Lists

- Since lists are mutable (i.e., can be changed), it is often useful to copy a list before performing operations on it

```
> numbers_copy = numbers[:]
> numbers_copy[1] = 3
> print(numbers, numbers_copy)
[2, 4, 6, 8, 10] [2, 3, 6, 8, 10]
```

Assigning Multiple List Elements

- You can also use slices to assign multiple list elements at the same time:

```
> numbers = [2, 4, 6, 8, 10]
> numbers[1:3] = [3, 5]
> print(numbers)
[2, 3, 5, 8, 10]
```

Adding an Element to a List

- Python provides a function called *append* to add an element to a list
  - E.g.

```
> numbers = [2, 4]
> numbers.append(6)
> print(numbers)
[2, 4, 6]
```

Adding Elements to a List

- Python provides a function called *extend* to add all elements in a list to a list
  - E.g.

```
> numbers = [2, 4]
> numbers.extend([6, 8])
> print(numbers)
[2, 4, 6, 8]
```

- You can also write this as `numbers + [6, 8]`
  - How is this different to `numbers.append([6, 8])`?

Sorting List Elements

- Python provides a function called *sort* to alter the order of elements in a list from lowest to highest
  - E.g.

```
> numbers = [6, 8, 4, 2]
> numbers.sort()
> print(numbers)
[2, 4, 6, 8]
```

- Note `numbers = numbers.sort()` won't do what we want

Deleting Elements from a List

- If you know the index of the element to delete, you can use *pop*

```
> test = ['a', 'b', 'c']
> deleted_element = test.pop(1)
> print(deleted_element, test)
'b' ['a', 'c']
```

- If you know the value of the element to delete, you can use *remove*

```
> test = ['a', 'b', 'c']
test.remove('b')
> print(test)
['a', 'c']
```

Summary

- Lists are sequences of values
  - They can be assigned to variables
- Access list elements by starting with index 0
- Use for loops to go through each element of a list
- List operations and functions make lists more useful

Outline

- Functions
  - Return values
  - Incremental development
  - Composition

Function Calls

- We have already seen some of Python's built-in functions:

```
type(2)
int('44')
test.sort()
```

- But what is a function, and how does it work?

Functions

- A function is a named sequence of statements that performs a computation
- Just like placing commands in a script file makes it easier for us to reuse the script, functions make it easier for us to reuse smaller portions of code

Function Characteristics

- Each function has:
  - A name
  - A set of arguments
  - A return value
- E.g., `type(31)`
  - The function name is *type*
  - The set of arguments is 31
  - The return value is <*class 'int'*>

Maths Functions

- Python has a built-in *math* module that has many mathematical functions
  - A module is a file that contains a collection of related functions
- To use the functions in a module, you first have to import the module with the *import* statement

```
import math
```

- This creates a module object named math
- To use a function in a module, you specify the module's name and the function's name, separated by a . (called *dot notation)*
  - E.g. `math.sqrt(2)`

Other Maths Functions

- `math.log10(100)`
- `math.log(100)`
- `math.sin(90 / 180 * math.pi)`
- `math.ceil(3.7)`
- `math.factorial(3)`

- See https://docs.python.org/3/library/math.html for a complete list

Creating Functions

- We know how to use built-in functions, but how do we create our own?
- A function definition specifies the name of a new function and the sequence of statements to run when it is called
- E.g.,

```
def print_numbers():
    print(1, 2, 3, 4, 5)
    print(6, 7, 8, 9, 10)
```

- Function names follow the same rules as variable names

Anatomy of a Function Definition

- Each function is made up of a header (the first line) and a body (everything else)
- The header specifies the name of the function and the arguments the function takes
  - If the function does not take any arguments, use ()
- The body is indented and can contain any number of statements

Calling a Function

- The syntax for using a function you define is the same as the syntax for in-built functions

```
print_numbers()
```

- A function must be defined before it is used
- When a function has been defined, it can be used anywhere a statement can go
  - Even in other functions

```
def print_twice():
  print_numbers()
  print_numbers()
```

Example

```
def print_numbers():
  print(1, 2, 3, 4, 5)
  print(6, 7, 8, 9, 10)

def print_twice():
  print_numbers()
  print_numbers()

print_twice()
```

Outputs:

```
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5
6 7 8 9 10
```

Function Arguments

- So far, the functions we have created don't take any arguments
- But we have seen some functions that do require arguments
  - E.g., `math.sin()`
- We specify the parameters to be passed to a function in the ()s
  - If more than one argument is required, separate them with commas

Function Arguments Example

```
def print_sum(a, b):
  sum = a + b
  print(sum)

print_sum(1, 1)
print_sum('hello', 'world')
```

Will output:

```
2
helloworld
```

Function Arguments Details

- Arguments can be passed positionally

```
print_sum(1, 1)
```

- Or by name

```
print_sum(b = 1, a = 1)
```

- Arguments can have default values

```
def print_sum(a, b = 1):
    sum = a + b
    print(sum)


print_sum(1)
print_sum(1, 2)
```

- The final parameter can also be a list to allow a variable number of arguments
  - Give it a name starting with * (e.g. def print_sum(a, b, *c))

Passing Variables

```
def print_sum(a, b):
  sum = a + b
  print(sum)
a = 1
b = 2
c = 3
print_sum(a, c)
print(b)
```

Will output:

```
4
2
```

Local Variables

- Variables and parameters defined in a function are only available inside that function
  - E.g. The following gives a NameError because `a` is not defined outside `print_sum`

```
def print_sum(a, b):
    print(a + b)
print(a)
```

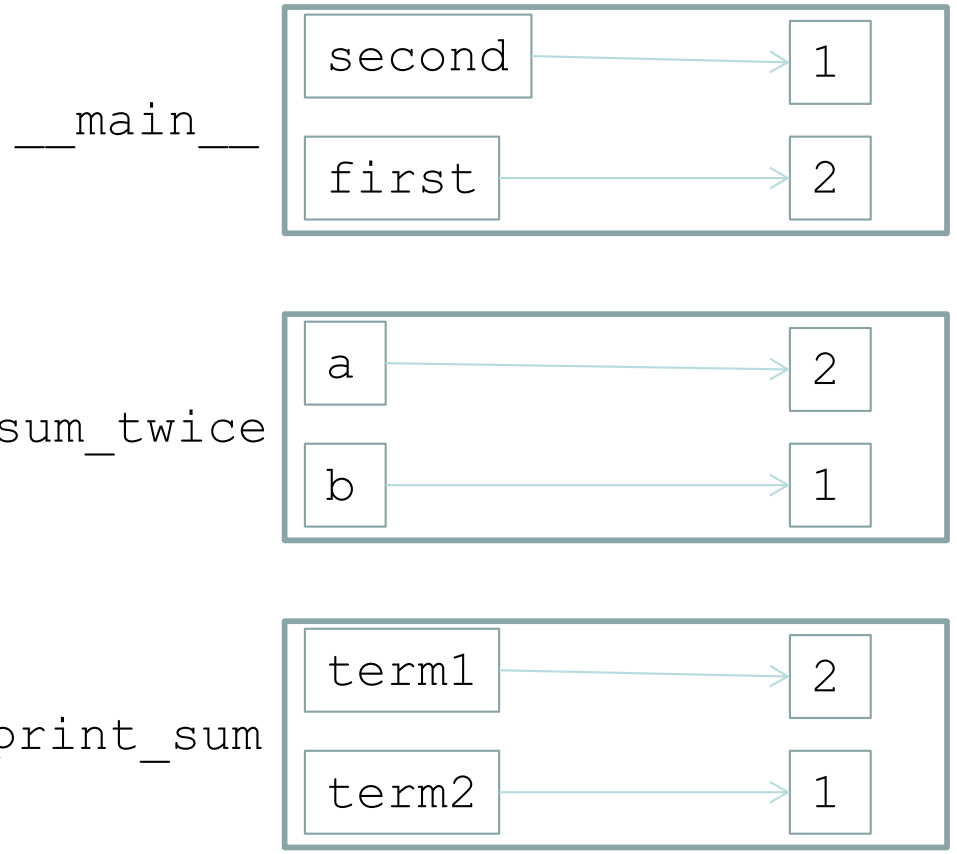
- `a` and `b` are given values when we call `print_sum` but they are destroyed as soon as the function completes

Stack Diagrams

- To help keep track of which variables are available to be used in different parts of a program, it can be useful to draw a stack diagram
  - Showing the function each variable belongs to and its value
- Each function is represented by a frame
  - A box with the name of the function beside it and the parameters and variables inside it
- The frames are arranged in a stack that shows which function called which

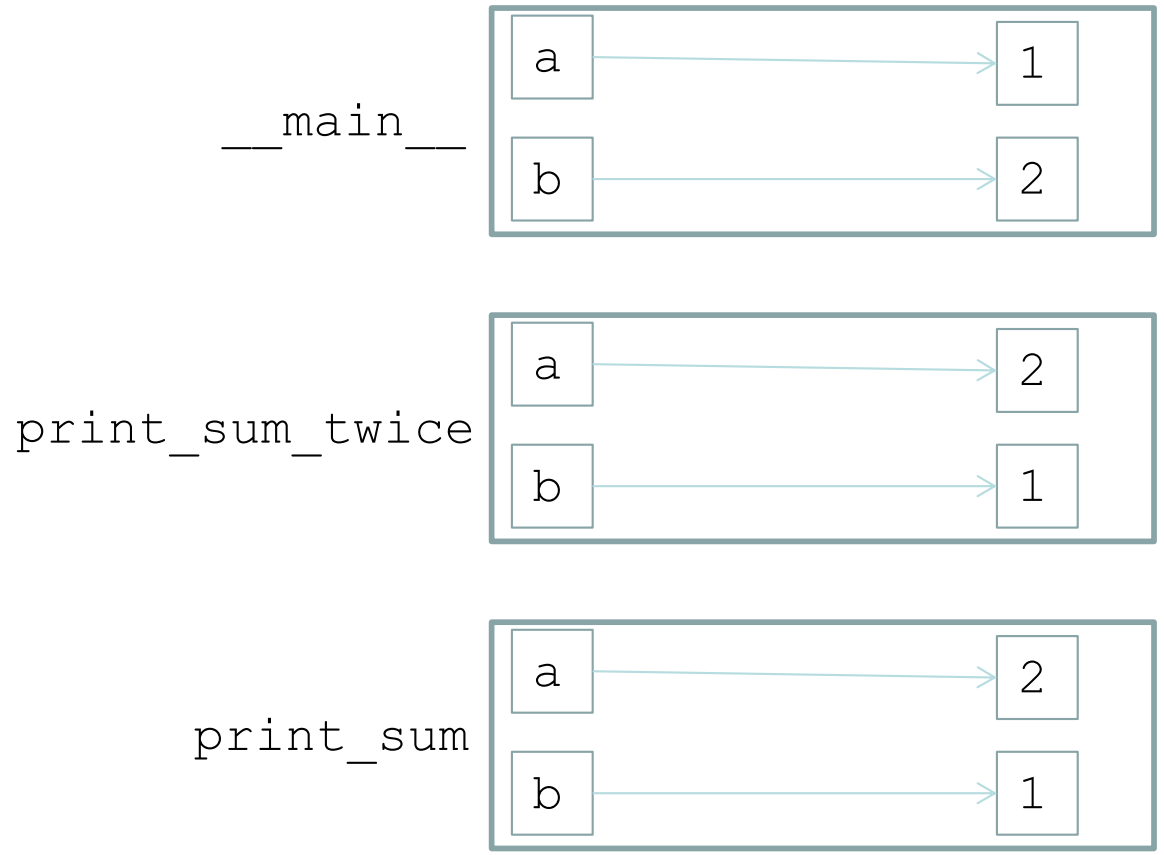
Stack Diagram Example

```
def print_sum(term1, term2):
  print(term1 + term2)
def print_sum_twice(a, b):
  print_sum(a, b)
  print_sum(a, b)
second = 1
first = 2
print_sum_twice(first,second)
```



Reusing Variable Names

```
def print_sum(a, b):
  print(a + b)
def print_sum_twice(a, b):
  print_sum(a, b)
  print_sum(a, b)
a = 1
b = 2
print_sum_twice(b,a)
```



Tracebacks

- If an error occurs during a function call, Python prints the name of the function that caused the error, the name of the function that called that function, the name of the function that called *that* function, all the way back to `__main__`
- This is called a *traceback*, and is useful for debugging
  - It tells you:
    - Which program file caused the error
    - Which functions were executing
    - Which line of code caused the problem

Traceback Example

```
def print_sum(term1, term2):
  print(term1 + b)
def print_sum_twice(a, b):
  print_sum(a, b)
  print_sum(a, b)
second = 1
first = 2
print_sum_twice(first,second)
```

Gives:

```
Traceback (most recent call last):
  File "sum.py", line 8, in __main__
    print_sum_twice(first, second)
  File "sum.py", line 4, in print_sum_twice
    print_sum(a, b)
  File "sum.py", line 2, in print_sum
    print(term1 + b)
NameError: name 'b' is not defined
```

Void Functions

- Some functions return results
  - E.g. `math.sqrt(4)`
- We can use that function anywhere the type of the value returned can be used
  - E.g. `x = 1 + math.sqrt(4)`
- So far, the functions we have defined do not return a value
  - These are called *void* functions

`a = print_sum(1, 2)` results in `a` having the value `None`

Return Values

- We can have our functions return values using the `return` statement
- E.g.,

```
def area(radius):
    a = math.pi * radius ** 2
    return a


circle_area = area(3)
```

- The value returned by the function is the value of the expression immediately after the `return` keyword

Returns

- A void function can use `return` statements too
  - It just has no expression after the return
- Other functions can return a value of any type
- A function can have multiple `return` statements
  - The function stops after the first one program flow reaches
- E.g.,

```
def absolute_value(x):
  if x < 0:
    return -x
  return x

absolute = absolute_value(-1)
value = absolute_value(1)
```

Recursion

- Functions can call other functions
  - They can even call themselves
  - This is called recursion
- E.g.,

```
def fibonacci(n):
  if n < 2:
    return 1
  return fibonacci(n - 1) + fibonacci(n - 2)
```

- You must be careful to ensure there is a base case
  - Otherwise the function will just keep calling itself

Summary

- Functions let us reuse code more easily
  - Break program up into more understandable pieces
- We can import modules to give us access to more functions
  - Or we can define our own
- Functions have arguments and local variables
  - Stack diagrams help us understand where they are available
- Functions can return a value
  - Or return None
- Recursion is an alternative to loops