CL02:
Introduction to Functions

Housekeeping

- First quiz on Thursday
- Seat assignments will be emailed tomorrow!
- Tutoring
  - In FB141 (location subject to change)
  - Monday, Wednesday, and Thursday 5-7 pm
- Office Hours
  - Usual times (check website)
- Practice Problems
  - On course site
- Review Session
  - FB009 and virtual (link will be on homepage)
  - Wednesday 6-8 pm
  - We will try to record it!

Functions

A function is a sub-program that defines what happens when a function is called.

Lets you generalize problems for different inputs

Help you *abstract away* from certain processes

Can be:

- Built-in
- Imported in Libraries
- DIY - Define in your python file

Abstraction Example

- Ordering a pizza…
  - You order a large cheese pizza
  - You don’t need to think about how they make the crust, got the ingredients, how long they bake it for, etc.
- round(x)
  - You round 10.25 down to 10 by calling round(10.25)
  - You don’t think about line by line how the some program is making this rounding decision

Calling a Function

Function Call: expressions that result in (“return”) a specific type

Common expressions:

- “Making a function call”
- “Using a function”
- “Invoking a function”

Looks like function_name(<inputs>)

E.g. print(“Hello”) , round(10.25), etc.

Examples…

print()

len()

randint()

Defining Functions

- So far we've only used built-in functions or ones imported from other libraries, but you can define your own as well!

- Allows you define solutions in one place of your program and reuse them in other places of your program file.. and even in other program files!

Function Syntax

Syntax for Calling A Built-In Function

function_name(<argument list>)

Syntax for Calling A Built-In Function

function_name(<argument list>)

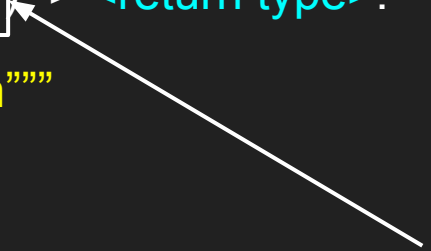
```
print(“hello”)
round(10.25)
randint(1,7)
randint(1,2+5)
```

Syntax for Defining A Function

```
def function_name(<parameter list>) -> <return type>:
    """Docstring describing function"""
    <what your function does>
```

Syntax for Defining A Function

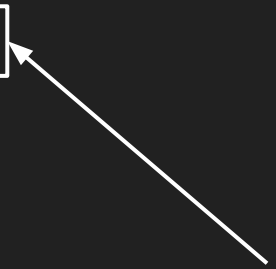
```
def function_name(<parameter list>) -> <return type>:
    """Docstring describing function"""
    <what your function does>
```



Generic inputs that you want your function to use (not specific values)

Syntax for Defining A Function

```
def function_name(<parameter list>) -> <return type>:
    """Docstring describing function"""
    <what your function does>
```



If your function *returns* something, this will be its type.
(You always return objects using the return keyword)

Syntax for Defining A Function

```
def function_name(<parameter list>) -> <return type>:
    """Docstring describing function"""
    <what your function does>
```

Practice: Write a function called sum
that takes two ints: num1 and num2 as inputs
and returns the **sum** of the two numbers.

function name

parameter list

return type

```
1   def sum(num1: int, num2: int) -> int:
2       """Add two numbers together."""
3       return num1 + num2
```

signature

```
1  def sum(num1: int, num2: int) -> int:
2      """Add two numbers together."""
3      return num1 + num2
```

Syntax for Calling A Defined Function

function_name(<parameter0> = <arg0>, <parameter1> = <arg1>, …)

sum(num1 = 11, num2 = 3)

Call vs. Signature

Signature (for defining a function) :

```
def function_name(<parameter list>) -> <return type>:
```

```
def sum(num1: int, num2: int) -> int:
```

Call (for calling a function):

```
function_name(<parameter0> = <arg0>, <parameter1> = <arg1>, …)
```

```
sum(num1 = 11, num2 = 3)
```

Call vs. Signature

```
def sum(num1: int, num2: int) -> int:


sum(num1 = 11, num2 = 3)
```

Call vs. Signature

```
def sum(num1: int, num2: int) -> int:
```

```
sum(num1 = 11, num2 = 3)
```

Call vs. Signature

```
def sum(num1: int, num2: int) -> int:
```

```
sum(num1 = 11, num2 = 3)
```

Call vs. Signature

```
def sum(num1: int, num2: int) -> int:
```

“parameters”

```
sum(num1 = 11, num2 = 3)
```

“arguments”

Call vs. Signature

```
def sum(num1: int, num2: int) -> int:
```

```
sum(num1 = 11, num2 = 3)
```

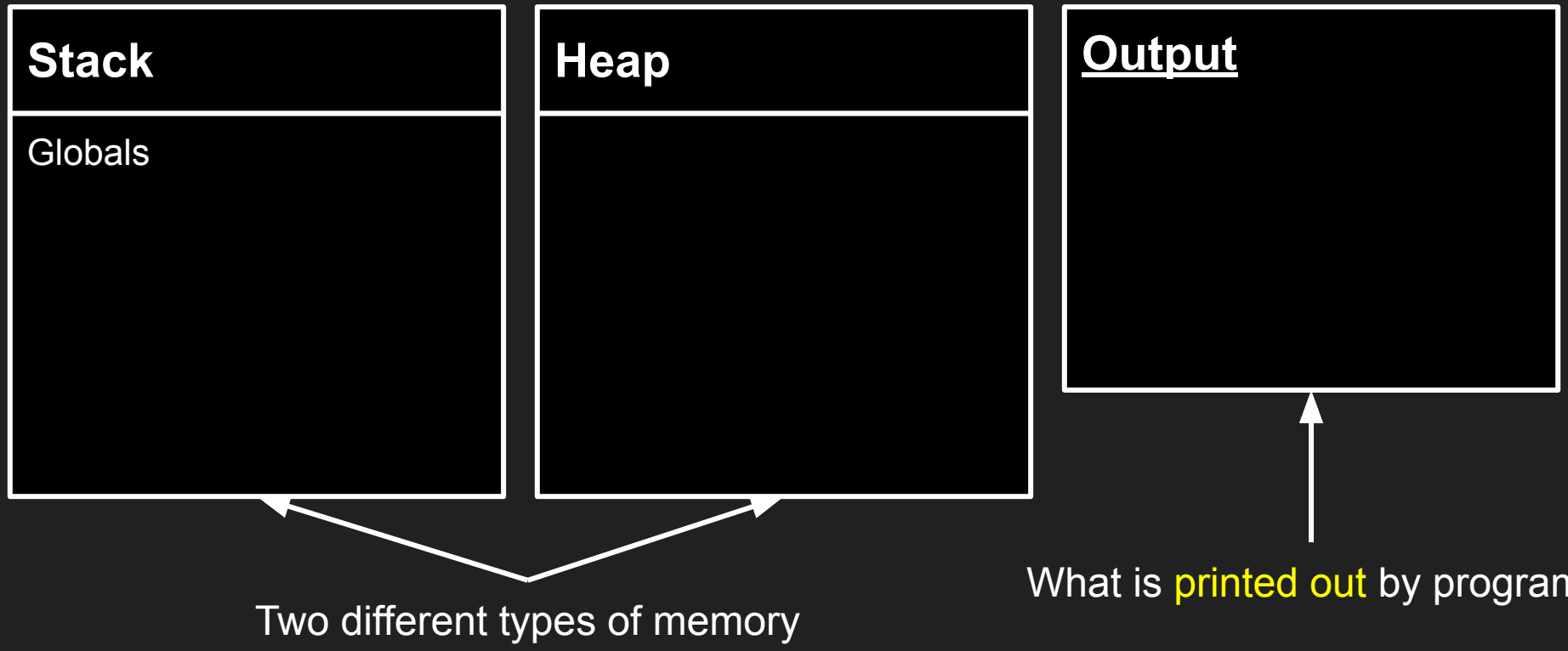
(evaluates to an int)

Memory Diagrams

Motivation

- Memory diagrams allow us to trace code in memory
- Helps us to understand what our code is doing and why

Memory Diagram Components



Stack vs. Heap

- Stack: variables, primitive types
- Heap: definitions, certain mutable types (more on this later)

```
1   def sum(num1: int, num2: int) -> int:
2       """Add two numbers together."""
3       return num1 + num2
4
5
6   print(sum(num1=4, num2=5))
```

Stack	Heap	Output
Globals		

Function Call Steps

- Prepare for call:
  - Has function been defined?
  - Are arguments fully evaluated?
  - Do parameters and arguments agree?
- Establish frame for function call:
  - Frame on stack labeled with function name
  - Return address
  - Copy over arguments

```python
def get_tax(price: int, tax_rate: float) -> float:
    return price * tax_rate

def total_cost(cost: int, tax: float) -> float:
    return cost + get_tax(price=cost, tax_rate=tax)

print(total_cost(cost=100, tax=0.07))
```

Stack	Heap	Output
Globals		