COMP 110

Expressions

Last Lecture

- Data Types
  - float (decimal, e.g. 2.0)
  - int (whole number, e.g. 2)
  - str (string of characters, e.g. “Hello”)
  - bool (evaluates to True or False e.g. True, 2 >= 3)
- Check type
  - type()
- Change type
  - str(), float(), int()

Expressions

- Something that *evaluates* at runtime
- Every expression evaluates to a specific typed value
- Examples
  - 1 + 2 * 3
  - 1
  - 1.0 * 2.0
  - “Hello” + “ World!”
  - 1 > 3

Numerical Operators

Operator Name	Symbol
Addition	+
Subtraction/Negation	-
Multiplication	*
Division	/
Exponentiation	**
Remainder “modulo”	%

Addition +

- If numerical objects, add the values together
  - 1 + 1 → 2
  - 1.0 + 2.0 → 3.0
- If strings, concatenate them
  - “Comp” + “110” → “Comp110”
- The result type depends on the operands
  - float + float → float
  - int + int → int
  - float + int → float
  - int + float → float
  - str + str → str

Addition +

- If numerical objects, add the values together
  - 1 + 1 → 2
  - 1.0 + 2.0 → 3.0
- If strings, concatenate them
  - “Comp” + “110” → “Comp110”
- The result type depends on the operands
  - float + float → float
  - int + int → int
  - float + int → float
  - int + float → float
  - str + str → str

Question: What happens when you try to add incompatible types?

Subtraction/Negation -

- Meant strictly for numerical types
  - 3 - 2 → 1
  - 4.0 - 2.0 → 2.0
  - 4.0 - 2 → 2.0
  - - (1 + 1) → -2
- The result type depends on the operands
  - float - float → float
  - int - int → int
  - float - int → float
  - int - float → float

Multiplication *

- If numerical objects, multiply the values
    - 1 * 1 → 1
    - 1.0 * 2.0 → 2.0
- If string and int, repeat the string
    - “Hello” * 3 → “HelloHelloHello”
- The result type depends on the operands
    - float * float → float
    - int * int → int
    - float * int → float
    - int * float → float
    - str * int → str

Division /

- Meant strictly for numerical types
  - 3 / 2 → 1.5
  - 4.0 / 2.0 → 2.0
  - 4 / 2 → 2.0
- Division results in a float
  - float / float → float
  - int / int → float
  - float / int → float
  - int / float → float

Exponentiation **

- Meant strictly for numerical types
  - 2 ** 2 → 4
  - 2.0 ** 2.0 → 4.0
- The result type depends on the operands
  - float ** float → float
  - int ** int → int
  - float ** int → float
  - int ** float → float

Remainder “modulo”

- Calculates the *remainder* when you divide two numbers
- Meant strictly for numerical types
  - 5 % 2 → 1
  - 6 % 3 → 0
- The result type depends on the operands
  - int % int → int
  - float % float → float
  - float % int → float
  - int % float → float
- Note:
  - If x is even, x % 2 → 0
  - If x is odd, x % 2 → 1

Order Of Operations

- P ()
- E **
- MD * / %
- AS + -
- Tie? Evaluate *Left to Right*

Relational Operators

Operator Name	Symbol
Equal?	==
Less than?	<
Greater than?	>
Less than or equal to? (At most)	<=
Greater than or equal to? (At least)	>=
Not equal?	!=

Relational Operators

- Always result in a bool (True or False)
- Equals (==) and Not Equal (!=)
  - Can be used for all primitive types we've learned so far! (bool, int, float, str)
- Every other type
  - Just use on floats and ints
  - (Can *technically* use on all primitive types)

Practice! Simplify and Type

- 2 + 4 / 2 * 2

- 220 >= int((“1” + “1” + “0”) * 2)

Simplify: $2 + 4 / 2 * 2$

(Reminder: P E M D A S)

Simplify: 2 + 4 / 2 * 2

What type is 2 + 4 / 2 * 2?

Simplify:

220 >= int((“1” + “1” + “0”) * 2)

Mods Practice! Simplify

- 7 % 2
- 8 % 4
- 7 % 4
- Any even number % 2
- Any odd number % 2

COMP 110

Introduction to Functions

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

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

signature

```python
def sum(num1: int, num2: int) -> int:
    """Add two numbers together."""
    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)

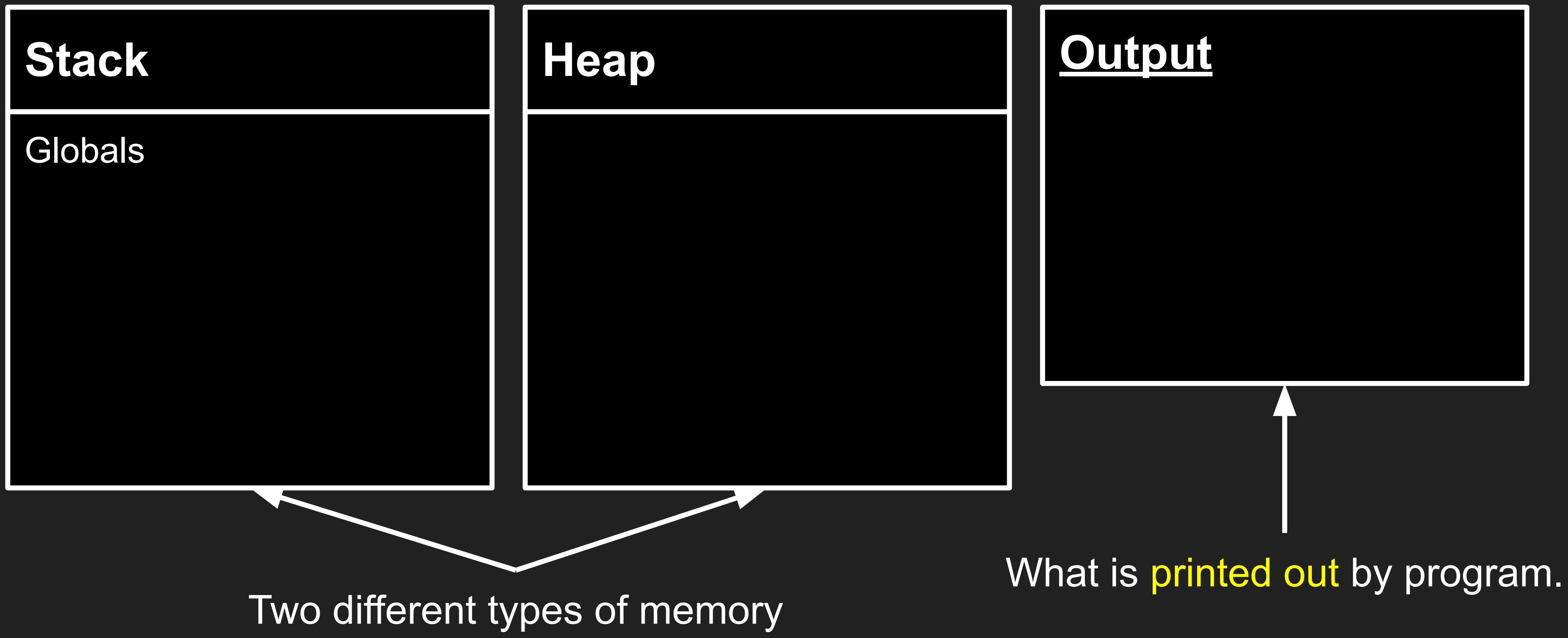
COMP 110

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)

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


print(sum(num1=4, num2=5))
```

Stack
Globals

Heap

Output

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

Globals

Heap

Output