Lecture 3: Control

CS 61A - Summer 2024

Charlotte Le & Laryn Qi

Announcements

Contents

1: Print & None

None Indicates that Nothing is Returned

The special value **None** represents nothing in Python

A function that does not explicitly return a value will return **None**

Careful: **None** is *not displayed* by the interpreter as the value of an expression

```
>>> def does_not_return_square(x):
...     x * x
...
>>> does_not_return_square(4)
>>> sixteen = does_not_return_square(4)
>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

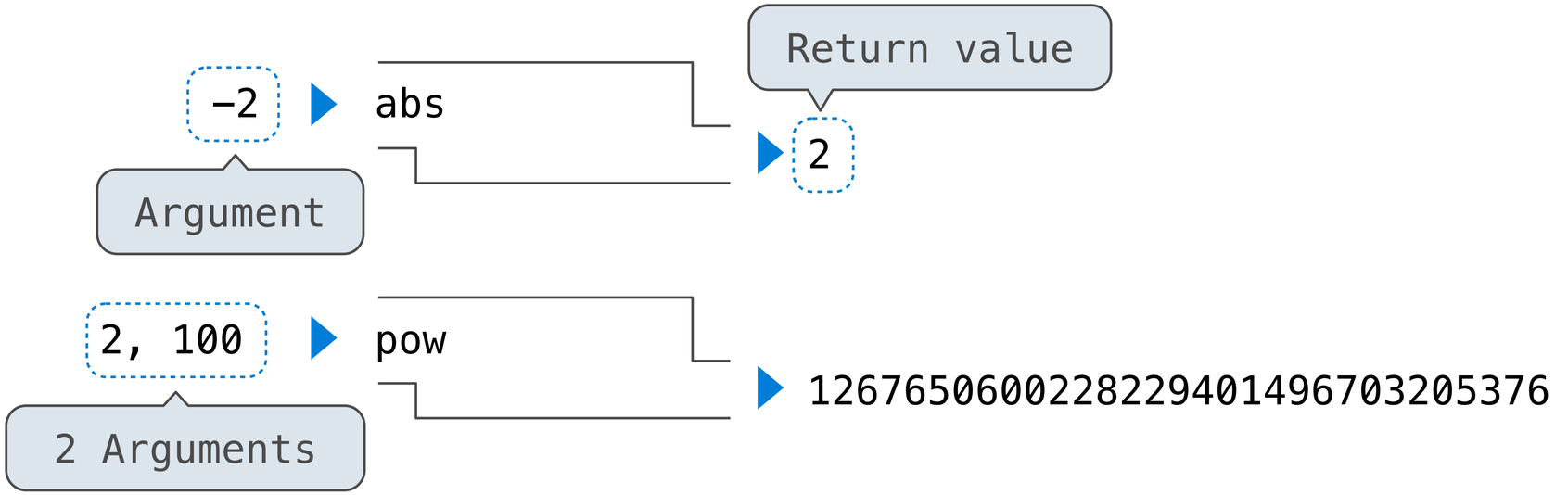
No **return**

None value is not displayed

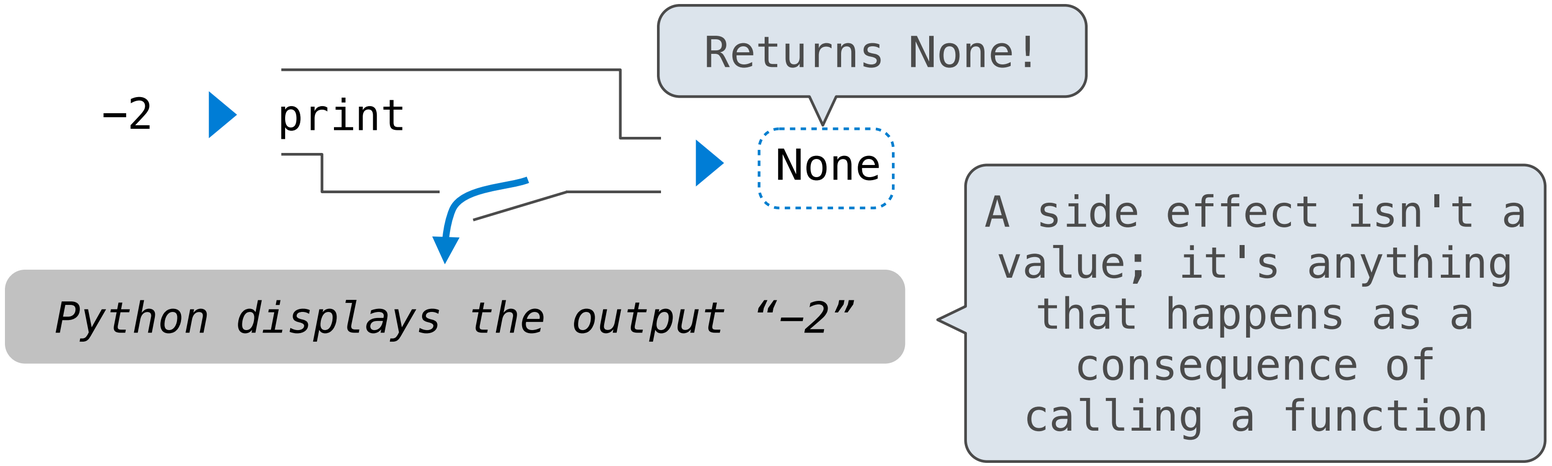
The name **sixteen** is now bound to the value **None**

Pure Functions & Non-Pure Functions

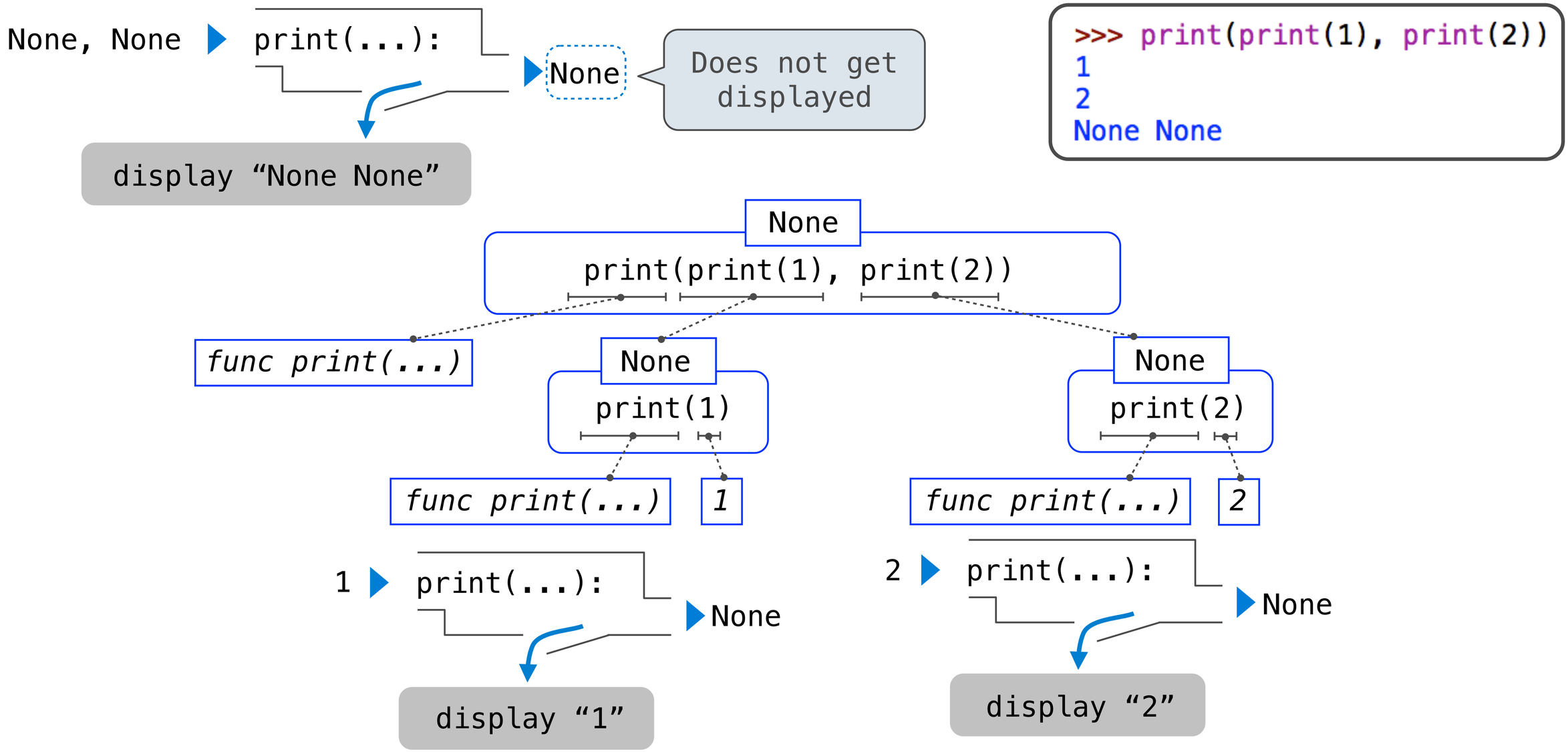
Pure Functions
just return values



Non-Pure Functions
have side effects



Nested Expressions with Print



2: Miscellaneous Python Features

Operators

```
>>> 2 + 3 * 4 + 5
19
>>> from operator import add, mul
>>> add(add(2, mul(3, 4)), 5)
19
>>> (2 + 3) * (4 + 5)
45
>>> mul(add(2, 3), add(4, 5))
45
```

Division

```
>>> 10 // 3 # floor division
3
>>> 10 / 3 # true division
3.3333333333333335
>>> 10 % 3 # modulus operator
1
>>> 27 ** (1/3) # exponentiation operator
3.0
```

Multiple Return Values

- We can assign multiple values to multiple variables in one statement.

```
>>> a, b, c = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3
```

```
>>> quotient, remainder = 10 // 3, 10 % 3
>>> quotient
3
>>> remainder
1
```

Multiple Return Values

- We can return multiple values from a function.
- First value: **a // b**
- Second value: **a % b**

```
>>> def divide_exact(a, b):
...     return a // b, a % b
...
>>> first, second = divide_exact(10, 3)
>>> first
3
>>> second
1
```

Interpreter vs. Code Editor

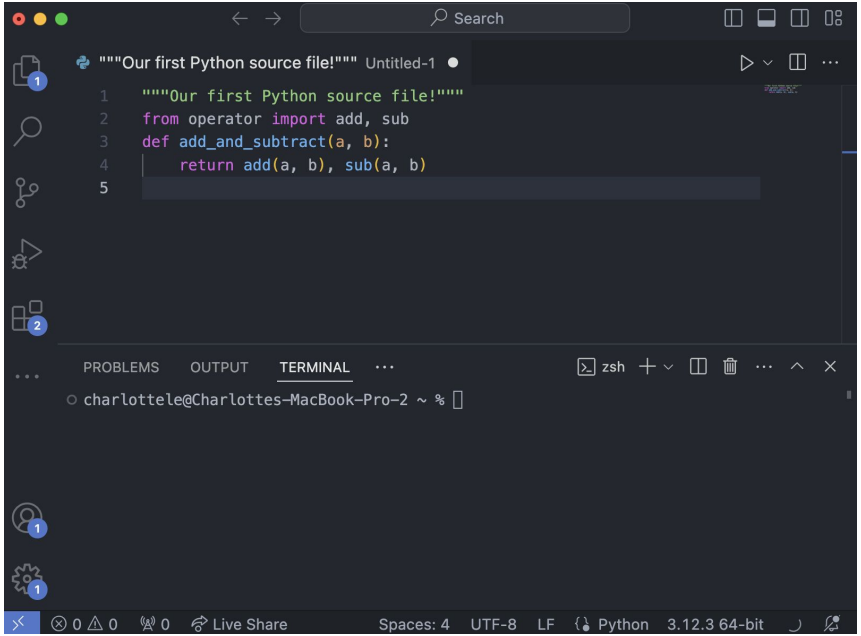
- The Python interpreter reads and executes code line-by-line.
- **python3 -i** starts the Python interpreter in interactive mode

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
charlottele@Charlottes-MacBook-Pro-2 ~ % python3 -i
Python 3.12.3 (main, Apr  9 2024, 08:09:14) [Clang 15.0.0 (clang-1500.1.0.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from operator import add, sub
>>> def add_and_subtract(a, b):
...     return add(a, b), sub(a, b)
...
>>> sum, difference = add_and_subtract(1, 6)
>>> sum
7
>>> difference
-5
>>>
```

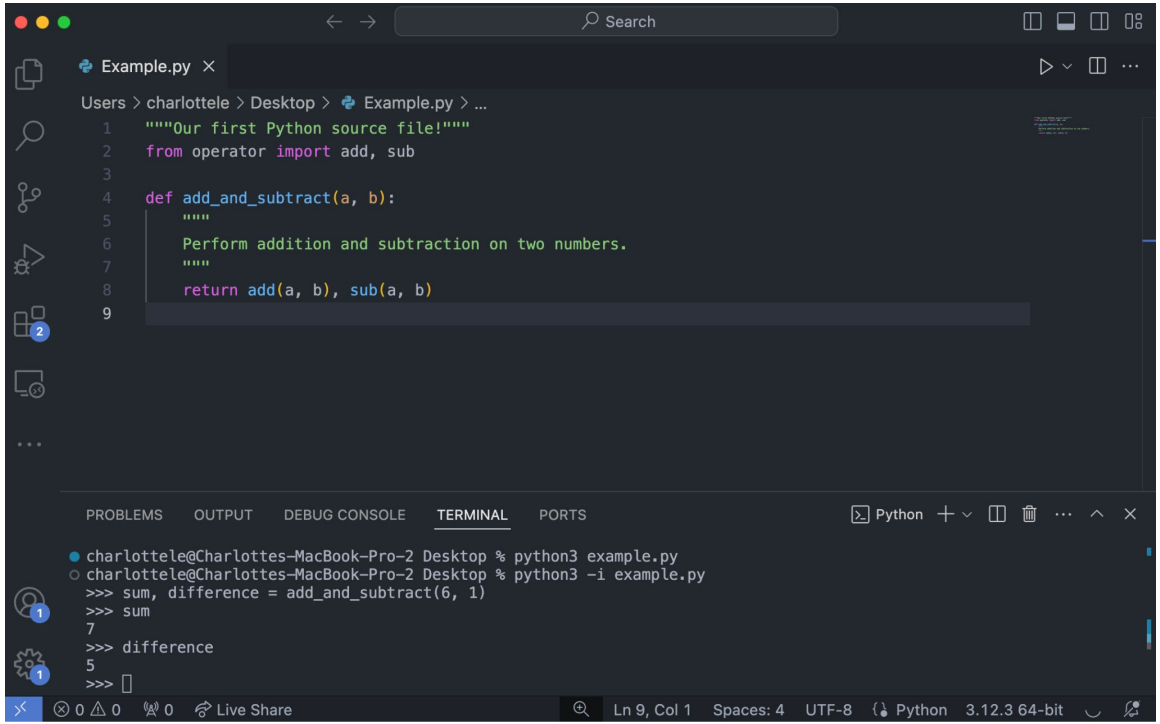
Interpreter vs. Code Editor

- The code editor allows us to write and edit code.



Interpreter vs. Code Editor

- **python3 -i *filename*.py** runs *filename*.py, executing its code
- After executing the script, you are now in a Python prompt (>>>) where you can interact with the environment created by the script



Interpreter vs. Code Editor

- **python3 -i *filename*.py** runs *filename*.py, executing its code
- After executing the script, you are now in a Python prompt (>>>) where you can interact with the environment created by the script

Docstrings

- A string used to document a Python function so we can understand what it does

Example.py

Users > charlottele > Desktop > Example.py > ...

```python
"""Our first Python source file!"""
from operator import add, sub

def add_and_subtract(a, b):
    """
    Perform addition and subtraction on two numbers.
    """
    return add(a, b), sub(a, b)
```

Doctests: Test Passed

- A way to include executable examples in the docstrings
- **python3 -m doctest -v *filename*.py** runs the doctests in *filename*.py and outputs the tests' results

Example.py

Users > charlottele > Desktop > Example.py > add_and_subtract

```python
"""Our first Python source file!"""
from operator import add, sub

def add_and_subtract(a, b):
    """
    Perform addition and subtraction on two numbers.
    >>> q, r = add_and_subtract(6, 1)
    >>> q
    7
    >>> r
    5
    """
    return add(a, b), sub(a, b)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
charlottele@Charlottes-MacBook-Pro-2 ~ % cd Desktop
charlottele@Charlottes-MacBook-Pro-2 Desktop % python3 -m doctest -v Example.py
Trying:
    q, r = add_and_subtract(6, 1)
Expecting nothing
ok
Trying:
    q
Expecting:
    7
ok
Trying:
    r
Expecting:
    5
ok
1 items had no tests:
    Example
1 items passed all tests:
   3 tests in Example.add_and_subtract
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
charlottele@Charlottes-MacBook-Pro-2 Desktop %
```

0 0 0 Live Share Ln 10, Col 10 Spaces: 4 UTF-8 LF Python 3.12.3 64-bit

Doctests: Test Failed

- A way to include executable examples in the docstrings
- **python3 -m doctest -v *filename*.py** runs the doctests in *filename*.py and outputs the tests' results

Example.py

Users > charlottele > Desktop > Example.py > ...

```python
"""Our first Python source file!"""
from operator import add, sub

def add_and_subtract(a, b):
    """
    Perform addition and subtraction on two numbers.
    >>> q, r = add_and_subtract(6, 1)
    >>> q
    7
    >>> r
    5
    """
    return add(a, b), add(a, b)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
charlottele@Charlottes-MacBook-Pro-2 Desktop % python3 -m doctest -v Example.py
Trying:
    q, r = add_and_subtract(6, 1)
Expecting nothing
ok
Trying:
    q
Expecting:
    7
ok
Trying:
    r
Expecting:
    5
**********************************************************************
File "/Users/charlottele/Desktop/Example.py", line 10, in Example.add_and_subtract
Failed example:
    r
Expected:
    5
Got:
    7
1 items had no tests:
    Example
**********************************************************************
1 items had failures:
   1 of   3 in Example.add_and_subtract
3 tests in 2 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
charlottele@Charlottes-MacBook-Pro-2 Desktop %
```

Default Arguments

- Values that the function argument will take if input is passed during the function call

Example.py

Users > charlottele > Desktop > Example.py > ...

```python
"""Demonstrating the power of default arguments"""
def calc_area(length=1, width=1):
    """
    Calculates the area of a rectangle.

    This function returns the area of a rectangle given its length and width.
    If no arguments are provided, the default length and width are 1.

    >>> calc_area()
    1
    >>> calc_area(5, 4)
    20
    >>> calc_area(3)
    3
    """
    return length * width
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
charlottele@Charlottes-MacBook-Pro-2 ~ % cd Desktop
charlottele@Charlottes-MacBook-Pro-2 Desktop % python3 -m doctest -v Example.py
Trying:
    calc_area()
Expecting:
    1
ok
Trying:
    calc_area(5, 4)
Expecting:
    20
ok
Trying:
    calc_area(3)
Expecting:
    3
ok
1 items had no tests:
    Example
1 items passed all tests:
   3 tests in Example.calc_area
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
charlottele@Charlottes-MacBook-Pro-2 Desktop %
```

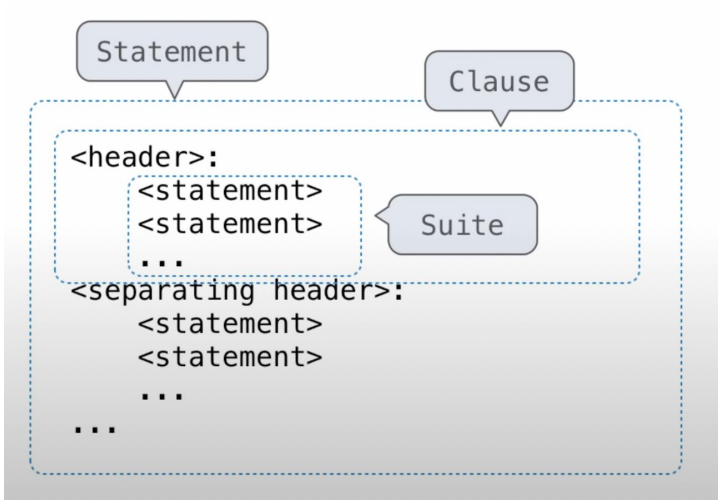
Ln 1, Col 51 Spaces: 4 UTF-8 LF Python 3.12.3 64-bit

Break

3: Control Structures (Conditional)

Statements

- A **statement** is executed by the interpreter to perform an action
- A **suite** is a sequence of statements
- A **clause** consists of a header and an indented suite of statements




```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

1 statement,
3 clauses,
3 headers,
3 suites

- To “execute” a suite means to execute its sequence of statements in order

Conditional Statements

```
if <conditional expression>:
    <suite of statements>
elif <conditional expression>:
    <suite of statements>
else:
    <suite of statements>
```

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

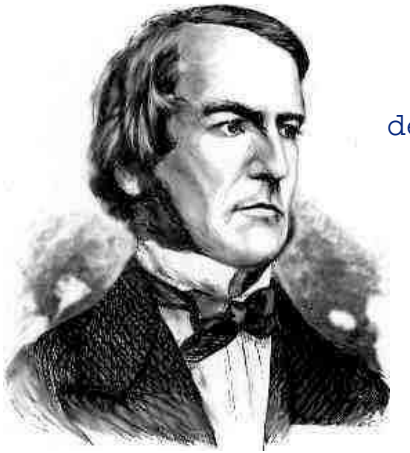
```
>>> absolute_value(-9)
9
>>> absolute_value(0)
0
>>> absolute_value(41)
41
```

Rules

1. **if** marks the start of an *if-elif-else* block
2. **elif** and **else** are optional
3. **if** and **elif** must have a conditional expression
4. an *if-elif-else* block can have multiple **elif** but only one **else**
5. each conditional expression is considered in order until a “truthy” value is reached
6. if a “truthy” value is reached, execute the suite then skip over all the rest of the *if-elif-else* block

4: Booleans

Boolean Contexts



George Boole

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

Two boolean contexts

“falsey” values: **False, 0, '', None** *(more to come)*

“truthy” values: **Everything else**

Boolean Operators

- **not**
  - returns the opposite boolean value of an expression
  - will always return either **True** or **False**

- **and**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *falsey* value and returns it
  - if all values evaluate to a *truthy* value, the last value is returned

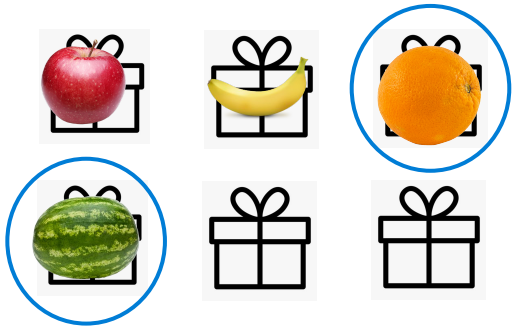
- **or**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *truthy* value and returns it
  - if all values evaluate to a *falsey* value, the last value is returned

```
>>> not True
False
>>> not None
True
>>> not -8
False
>>> not ""
True
>>> -1 and 0 and 1
0
>>> False or 9999 or 1/0
9999
>>> "i" and "love" and "cs" and "61a"
'61a'
```

Boolean Operators - Short Circuiting Example

and

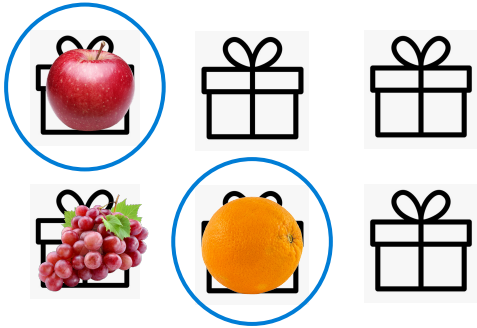
you guess the boxes contain:
an apple **and** a banana **and** an orange



- short-circuits at the first falsey value and returns it
- if all values evaluate to a truthy value, the last value is returned

or

you guess the boxes contain:
an apple **or** a banana **or** an orange



- short-circuits at the first truthy value and returns it
- if all values evaluate to a falsey value, the last value is returned

Boolean Operators

- **not**
  - returns the opposite boolean value of an expression
  - will always return either **True** or **False**

- **and**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *falsey* value and returns it
  - if all values evaluate to a *truthy* value, the last value is returned

- **or**
  - evaluates expressions in order
  - stops evaluating (short-circuits) at the first *truthy* value and returns it
  - if all values evaluate to a *falsey* value, the last value is returned

```
>>> not True
False
>>> not None
True
>>> not -8
False
>>> not ""
True
>>> -1 and 0 and 1
0
>>> False or 9999 or 1/0
9999
>>> "i" and "love" and "cs" and "61a"
'61a'
```

Boolean Operators: Be Careful!

- Order of operations: **not** > **and** > **or**

```
>>> True and not False or not True and False
True
>>> (True and (not False)) or ((not True) and False)
True
```

- **Error** != **False**
  - **False** is a value
  - **Error** is when something wrong with your code and it can't finish running (e.g., **ZeroDivisionError**, **SyntaxError**, **TypeError**, **NameError…**)

Boolean Operators - Short Circuiting in a Conditional Statement Example

```python
def bouncer(age, has_valid_id):
    """
    Two conditions must both be met before you let someone in.
    1: They must be at least 21 years old (age >= 21)
    2: They must have a valid ID (has_valid_id)
    """
    if age >= 21 and has_valid_id:
        print("Welcome to the club!")
    else:
        print("Sorry, you can't come in.")

bouncer(16, True)
# fails condition 1, so Python skips checking has_valid_id
```

```python
def bouncer(is_vip, has_valid_id):
    """
    Two conditions, but only one needs to be met to let someone in.
    1: They must be a VIP member (is_vip)
    2: They must have a valid ID (has_valid_id)
    """
    if is_vip or has_valid_id:
        print("Welcome to the club!")
    else:
        print("Sorry, you can't come in.")


bouncer(True, False)
# passes condition 1, so Python skips checking has_valid_id
```

5: Control Structures (Iteration)

While Statements

```
1  i, total = 0, 0
2  while i < 3:
3      i = i + 1
4      total = total + i
```

Global frame

i	~~1~~ ~~2~~ ~~3~~ 3
total	~~1~~ ~~3~~ ~~6~~ 6

Execution Rule for While Statements:

1. Evaluate the header's expression.

2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

George Boole

Example: `max_digit`

Summary

- None is a special value is not displayed by the interpreter as the value of an input expression
- `print` *returns* None and displays its argument as a *side-effect*
- **Docstrings** describe the behavior of a function in natural language
- **Doctests** describe the behavior of a function by providing input-output pairs
- **Booleans** are used to assign either True/False as the value of an expression
- **Conditional Statements** allow you to execute lines of code and skip others depending on the value of a boolean expression
- **While Statements** allow you repeat a block of code until a condition is met