Structure and Interpretation of Computer Programs with Python

Lesson 3

-Presented By: Sean Li

Quick Review

Call Expression:

- Components: Operator & Operand
- Evaluation Process:
  - From left to right
  - First evaluates the operator, then the operand
  - Apply the function that is the value of the operator to the arguments that are the values of the operands

Quick Review

Types of Expressions:

Types of Expressions

Primitive Expression: 2, add, "hello"

Number or Numeral Name String

Call Expression: add(1, 2)

Assignment (not that assignment)

Any difference?

```
a = 1
a == 1
```

Assignment (not that assignment)

Any difference?

a = 1 assignment

a == 1 equality operator

Assignment is a simple means of abstraction: binds names to values

Define a Function:

Function definition is a more powerful means of abstraction: binds names to expressions

```
def <name>(<formal parameters>):
    return <return expression>
```

Define a Function:

Function name

Function signature

```
def <name>(<formal parameters>):
    return <return expression>
```

Function body

Define a Function:

Function name: the name of the function

Function signature: the arguments that the function takes

```
def <name>(<formal parameters>):
    return <return expression>
```

Function body: defines the computation performed when the function is applied

Indentation:

Python uses indentation to determine the hierarchy of each level.

Demo

Return:

A return statement completes the evaluation of a call expression and provides its value

In other words, a function always ends with a return statement

Question:

If a function always ends with a return statement, then why don't I see a return statement sometimes?

Print and none:

None: Indicates that Nothing is Returned

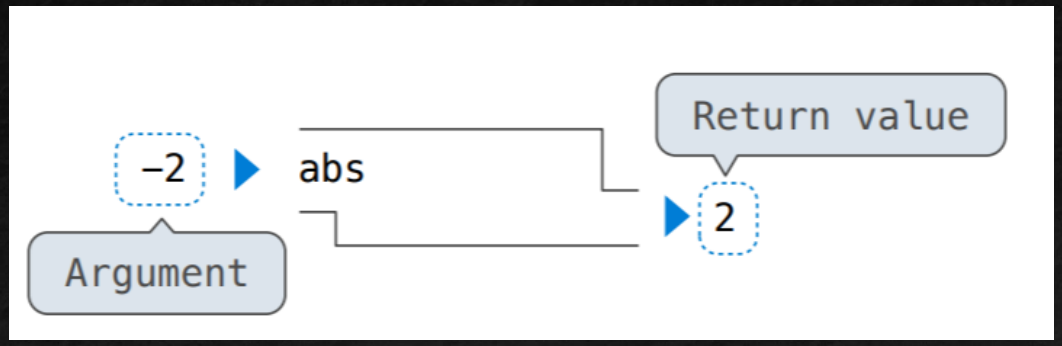
- A special value "None" represents nothing in Python
- A function that does not explicitly return a value will return None
- None is not displayed by the interpreter as the value of an expression

Demo:

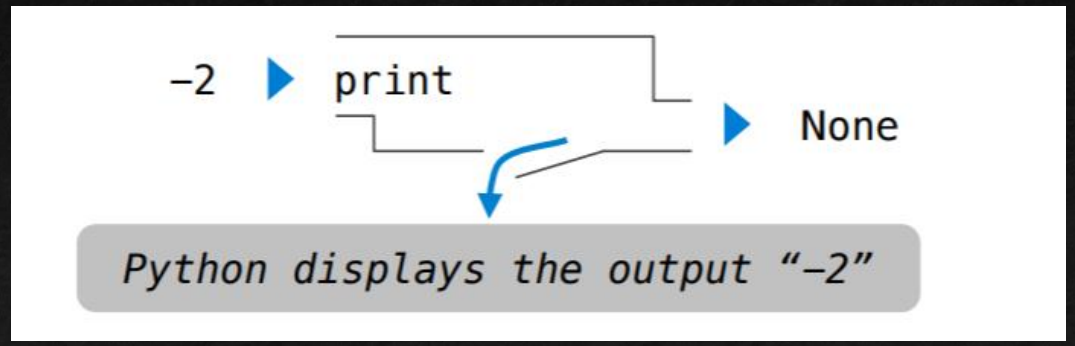
```python
def add(a, b):
    result = a + b
    return result

def add(a, b):
    result = a + b

```

Pure functions vs Non-pure functions:

Pure: has no side effects

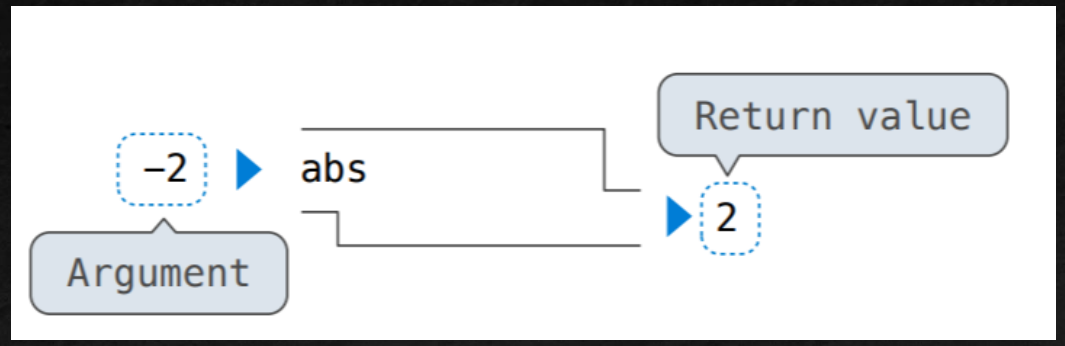


Non-Pure: has side effects

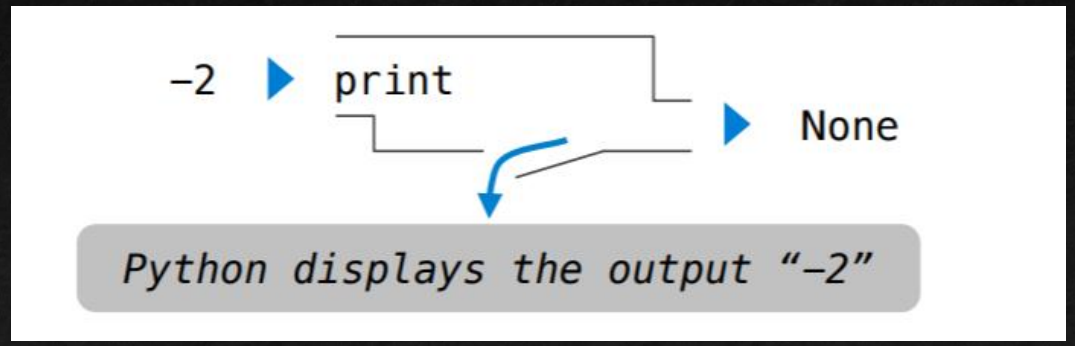


Pure functions vs Non-pure functions:

Pure: has no side effects



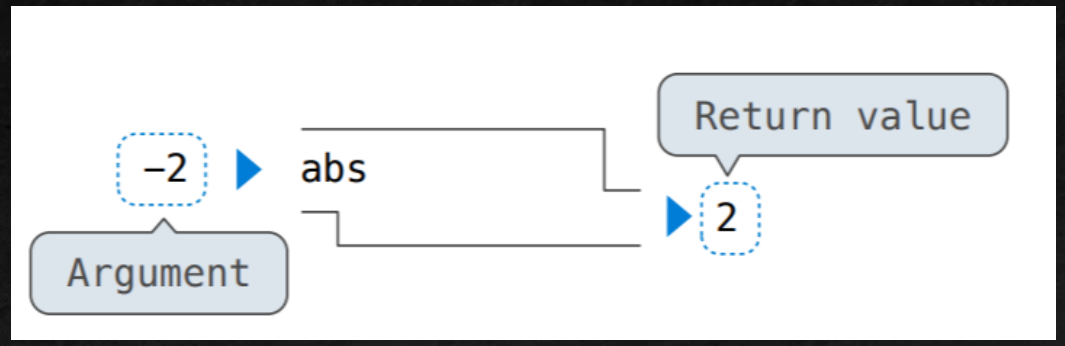
Non-Pure: has side effects



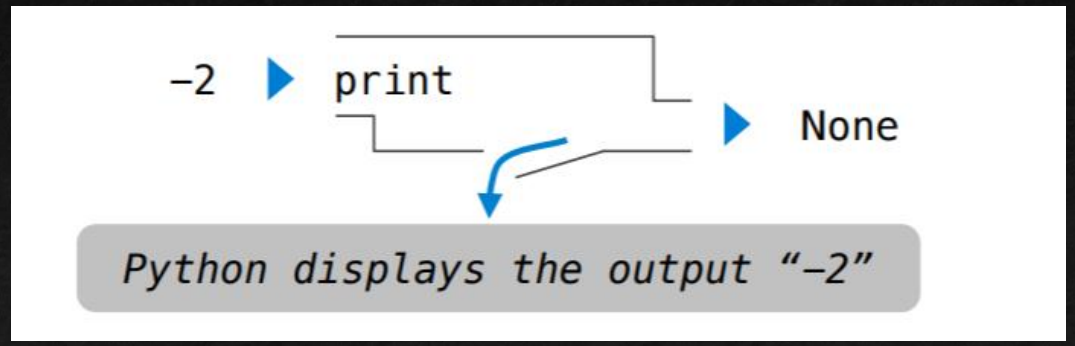
A side effect isn't a value; it's anything that happens as a consequence of calling a function

Pure functions vs Non-pure functions:

Pure: has no side effects



Non-Pure: has side effects



A side effect isn't a value; it's anything that happens as a consequence of calling a function

Output?

```
print(add(1, 2))
```

Output?

```
print(print("hello"), print("what"))
```

Life cycle of a user-defined function

Def statement:

```
def add(a, b):
    result = a + b
    return result
```

Name

Parameters/Input

Body

What happens: A new function is created! Name bound to that function in the current frame

Life cycle of a user-defined function

Call Expression: `add(1, 2)`

Operands

Operator: func add

What happens: Operator & operands get evaluated, Function (value of operator) gets called on arguments (values of operands)

Life cycle of a user-defined function

Calling:

What happens: This is the step where Python actually calls the function. A new frame is created! Parameters bound to arguments. Body is executed in that new environment

Life cycle of a user-defined function

Calling:

What happens: This is the step where Python actually calls the function. A new frame is created! Parameters bound to arguments. Body is executed in that new environment

Environment

- Every expression is evaluated in the context of an environment.
- A call expression and the body of the function being called are evaluated in different environments

demo

```python
def add(a, b):
    result = a + b
    return result

add(add(1, 2), 2)
```

If statement

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

If statement

- Only one of them will happen
- Can have as many elif as possible
- Does not necessarily have to end with else, but it's always good to add one

Example

You made a deal with your mom. If you get an A, you get a new toy. Otherwise, you get one more hour of study.

```
def deal_with_mom(A):
"""A is a boolean value that evaluates to either True or False"""
```

Any difference?

```python
if x < 0:
    print("yay")
else:
    print("omg")
```

```python
if x < 0:
    print("yay")
if x < 1:
    print("what?")
else:
    print("omg")
```

```python
if x < 0:
    print("yay")
elif x < 1:
    print("what?")
else:
    print("omg")
```

Any difference? (x = -1)

```python
if x < 0:
    print("yay")
else:
    print("omg")
```

```python
if x < 0:
    print("yay")
if x < 1:
    print("what?")
else:
    print("omg")
```

```python
if x < 0:
    print("yay")
elif x < 1:
    print("what?")
else:
    print("omg")
```

Exercise:

```
>>> def xk(c, d):
...     if c == 4:
...         return 6
...     elif d >= 4:
...         return 6 + 7 + c
...     else:
...         return 25
>>> xk(10, 10)
------

>>> xk(10, 6)
------

>>> xk(4, 6)
------

>>> xk(0, 0)
------
```

Exercise:

```
>>> def how_big(x):
...     if x > 10:
...         print('huge')
...     elif x > 5:
...         return 'big'
...     elif x > 0:
...         print('small')
...     else:
...         print("nothin'")
>>> how_big(7)
------

>>> how_big(12)
------

>>> how_big(1)
------

>>> how_big(-1)
------
```