Adv. Functions

David H Smith IV

University of Illinois Urbana-Champaign

Tues, Nov 16 2021

Course Overview

Reminders

- Things are due. Check the calendar.

Bytecode: Looking under the hood

Python code:

```
def hello():
    print("Hello, World!")
```

Bytecode:

```
0 LOAD_GLOBAL              0 (print)
2 LOAD_CONST               1 ('Hello, World!')
4 CALL_FUNCTION            1
6 POP_TOP
8 LOAD_CONST               0 (None)
0 RETURN_VALUE
```

- The computer doesn't just read the code you write.

Bytecode: Looking under the hood

Python code:

```
def hello():
    print("Hello, World!")
```

Bytecode:

```
0 LOAD_GLOBAL              0 (print)
2 LOAD_CONST               1 ('Hello, World!')
4 CALL_FUNCTION            1
6 POP_TOP
8 LOAD_CONST               0 (None)
0 RETURN_VALUE
```

- A The computer doesn't just read the code you write.
- B Python is a (kinda) interpreted language: source.py → bytecode.pyc → interpreter

Bytecode: Looking under the hood

Python code:

```
def hello():
    print("Hello, World!")
```

Bytecode:

```
0 LOAD_GLOBAL              0 (print)
2 LOAD_CONST               1 ('Hello, World!')
4 CALL_FUNCTION            1
6 POP_TOP
8 LOAD_CONST               0 (None)
0 RETURN_VALUE
```

- A The computer doesn't just read the code you write.
- B Python is a (kinda) interpreted language: source.py → bytecode.pyc → interpreter

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

A. Compiled vs Interpreted:

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

A. Compiled vs Interpreted:
   1. **compiled** → Convert source code into another language. Typically, though not exclusively, a higher level language (e.g., Python) into a lower level language (e.g., bytecode).

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

A. Compiled vs Interpreted:
   1. **compiled** → Convert source code into another language. Typically, though not exclusively, a higher level language (e.g., Python) into a lower level language (e.g., bytecode).
   2. **interpreted** → Another program reads the code you wrote one line at a time and performs those operations.

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

A. Compiled vs Interpreted:
   1. **compiled** → Convert source code into another language. Typically, though not exclusively, a higher level language (e.g., Python) into a lower level language (e.g., bytecode).
   2. **interpreted** → Another program reads the code you wrote one line at a time and performs those operations.

B. There's generally crossover between these.

Bytecode: Looking under the hood

You can view the bytecode of any function via the following:

```
import dis

def hello():
  print("Hello, World!")

dis.dis(hello)
```

A. Compiled vs Interpreted:
   1. **compiled** → Convert source code into another language. Typically, though not exclusively, a higher level language (e.g., Python) into a lower level language (e.g., bytecode).
   2. **interpreted** → Another program reads the code you wrote one line at a time and performs those operations.

B. There's generally crossover between these.

C. Why bother? Because interpreting bytecode is faster.

Functions are Objects

Poll Question: Functions

What, if anything, gets printed to the screen after this code executes?

```
def add1(x): return x + 1
def mul2(x): return x * 2

x = 1
fns = [add1, mul2, mul2, print]
for f in fns:
  x = f(x)
```

A. 1
B. 4
C. 8
D. SyntaxError

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` $\rightarrow$ The docstring for the function.

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` → The docstring for the function.
- `foo.__code__` → Gives the address of the code portion of the foo object in memory.

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` → The docstring for the function.
- `foo.__code__` → Gives the address of the code portion of the foo object in memory.
  - `foo.__code__.co_argcount` → The number of arguments in the function object.

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` → The docstring for the function.
- `foo.__code__` → Gives the address of the code portion of the foo object in memory.
  - `foo.__code__.co_argcount` → The number of arguments in the function object.
  - `foo.__code__.co_consts` → The literals present in the function object.

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` → The docstring for the function.
- `foo.__code__` → Gives the address of the code portion of the foo object in memory.
  - `foo.__code__.co_argcount` → The number of arguments in the function object.
  - `foo.__code__.co_consts` → The literals present in the function object.
  - `foo.__code__.co_name` → The name of the function.

Functions are Objects

```
def foo(arg1, arg2):
  if arg1 == arg2:
    return "They're equal"
  return "They're not equal"
```

- `foo.__doc__` → The docstring for the function.
- `foo.__code__` → Gives the address of the code portion of the foo object in memory.
  - `foo.__code__.co_argcount` → The number of arguments in the function object.
  - `foo.__code__.co_consts` → The literals present in the function object.
  - `foo.__code__.co_name` → The name of the function.
  - `foo.__code__.co_varnames` → A tuple of the names of variables present in the function object.

Namespace

Namespace

```
print('Initial global namespace: ')
print(globals())

my_var = "This is a variable"
print('\nCreated new variable')
print(globals())

def my_func():
    pass

print('\nCreated new function')
print(globals())
```

```
Initial global namespace:
{}

Created new variable
{'my_var': 'This is a variable'}

Created new function
{'my_func': <function my_func at 0x2349d4>,
      'my_var': 'This is a variable'}
```

1. Maps names to objects.

Namespace

```
print('Initial global namespace: ')
print(globals())

my_var = "This is a variable"
print('\nCreated new variable')
print(globals())

def my_func():
    pass

print('\nCreated new function')
print(globals())
```

```
Initial global namespace:
{}

Created new variable
{'my_var': 'This is a variable'}

Created new function
{'my_func': <function my_func at 0x2349d4>,
      'my_var': 'This is a variable'}
```

1. Maps names to objects.
2. You can check the local namespace with `locals()` and it returns a dictionary of names and values

Namespace

```
print('Initial global namespace: ')
print(globals())

my_var = "This is a variable"
print('\nCreated new variable')
print(globals())

def my_func():
    pass

print('\nCreated new function')
print(globals())
```

```
Initial global namespace:
{}

Created new variable
{'my_var': 'This is a variable'}

Created new function
{'my_func': <function my_func at 0x2349d4>,
      'my_var': 'This is a variable'}
```

1. Maps names to objects.
2. You can check the local namespace with `locals()` and it returns a dictionary of names and values
3. You can check the global namespace with `globals()`.

Namespace

```
print('Initial global namespace: ')
print(globals())

my_var = "This is a variable"
print('\nCreated new variable')
print(globals())

def my_func():
    pass

print('\nCreated new function')
print(globals())
```

```
Initial global namespace:
{}

Created new variable
{'my_var': 'This is a variable'}

Created new function
{'my_func': <function my_func at 0x2349d4>,
      'my_var': 'This is a variable'}
```

1. Maps names to objects.
2. You can check the local namespace with `locals()` and it returns a dictionary of names and values
3. You can check the global namespace with `globals()`.

Go to example 1.

Scope

Poll Question: Function Scoping

What is produced by the following code?

```
my_var = 11
def change_my_var():
  my_var = 12

change_my_var()
print(my_var)
```

A. 11
B. 12
C. NameError
D. None

Poll Question: Function Scoping

What is produced by the following code?

```
my_var = 11
def print_my_var():
  print(my_var)

print_my_var()
```

A. 11
B. 12
C. NameError
D. None

Poll Question: Function Scoping

What is produced by the following code?

```
my_var = 11
def print_my_var():
  print(my_var)

print_my_var()
```

A. 11
B. 12
C. NameError
D. None

1. We have read access but not write access in the function's scope.
2. How do we get write access to the global scope from within a function?

Poll Question: Function Scoping

What goes where the ?? is in order to (1) change the **global** value of my_var and (2) such that the user enters is printed to the screen when the code finishes running?

```
my_var = 11
def change_my_var(new_my_var):
  ??

change_my_var(int(input("Enter a new number: ")))
print(my_var)
```

Scope

- Namespaces and scopes go hand-in-hand.

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope $\rightarrow$ Contains all of the built-in's in Python (e.g., `int()`, `range()`)

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope → Contains all of the built-in's in Python (e.g., `int()`, `range()`)
   - Global Scope → Contains variables located outside of functions in the global scope.

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope → Contains all of the built-in's in Python (e.g., `int()`, `range()`)
   - Global Scope → Contains variables located outside of functions in the global scope.
   - Local Scope → The scope that is only accessible to a given function.

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope → Contains all of the built-in's in Python (e.g., `int()`, `range()`)
   - Global Scope → Contains variables located outside of functions in the global scope.
   - Local Scope → The scope that is only accessible to a given function.

C. Scope resolution → The processing searching a namespace for a variable.

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope → Contains all of the built-in's in Python (e.g., `int()`, `range()`)
   - Global Scope → Contains variables located outside of functions in the global scope.
   - Local Scope → The scope that is only accessible to a given function.

C. Scope resolution → The processing searching a namespace for a variable.

D. NameError → An error that's generated when scope resolution fails. In otherwords, the name isn't in global or local namespace.

Scope

A. Namespaces and scopes go hand-in-hand.

B. Types of scope:
   - Built-in Scope → Contains all of the built-in's in Python (e.g., `int()`, `range()`)
   - Global Scope → Contains variables located outside of functions in the global scope.
   - Local Scope → The scope that is only accessible to a given function.

C. Scope resolution → The processing searching a namespace for a variable.

D. NameError → An error that's generated when scope resolution fails. In otherwords, the name isn't in global or local namespace.

E. A function will search it's local namespace first then the global namespace.

Poll Question: Function Scoping

What is produced by the following code?

```
thing = 11
def find_var(my_var):
  return "thing" in local()

print(find_var("thing"))
```

A. True
B. False
C. SyntaxError
D. NameError

Functions

Function Returns

How many objects are returned by the following function?

```
def return_first_and_lst(a_list):
  return a_list[0], a_list[-1]
```

A. 1
B. 2
C. 3
D. None

Function Returns

How many objects are returned by the following function?

```
def return_first_and_lst(a_list):
  return a_list[0], a_list[-1]
```

A. 1
B. 2
C. 3
D. None

The above code is equivalent to this:

```
def return_first_and_lst(a_list):
  return (a_list[0], a_list[-1])
```

Poll Question: Function Args and Mutability

What is the value of `x[0]` after this code executes:

```
def remove_first(a_list):
  a_list = a_list[1:]

x = [1, 2, 3, 4]
remove_first(x)
```

A. 1
B. 2
C. 3
D. 4
E. Something else
F. An error occurs

Poll Question: Scope

What are the values of `w`, `x`, `y`, and `z` in the global scope after this code executes? What might we include in the line with the ?? to verify this?

```
x, y, z = (7, 5, 10)
def a_function(y):
  x = 2 * y
  return x * z

w = a_function(x)
??
```

A. `50, 7, 5, 10`

B. `100, 10, 5, 10`

C. `140, 7, 5, 10`

D. `140, 14, 5, 10`

E. SyntaxError