ECE326

PROGRAMMING LANGUAGES

Lecture 4b : Functions and Scope

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

None

- A value to signify “no value”
- Not the same as NULL in C++
  - No pointers in Python, only references
- Use identity operator to check for None

```
if var is None:
    ... handle situation ...
```

- Do NOT use equality operator
  - Can be overridden (operator overloading)
  - `is` cannot be overridden

Function

- A reusable sequence of program instructions
  - Usually has an associated name
  - Also known as subroutines
- In Python

```
def foo(parameters…):
    block
```

- Function can take zero or more parameters
- Return type does not need to be specified
  - Can return different types
  - returns `None` if function ends without `return`

Scope

- Name binding
  - Association of name to a variable, constant, or function
- Region of code where binding is valid
- Benefits
  - Helps prevent name collision
    - E.g. using variable name *foo* in two different functions
  - Allows same name to refer to different things
    - E.g. *foo* may be *int* in function *A*, and *float* in function *B*

Scope

- Lexical (static) scope
  - Used by most modern languages
  - Scope is the *program text* that encloses the name
    - E.g. a function variable’s scope is the function definition
  - Can be determined at compile time
- Dynamic scope
  - Scope is the time period that the enclosing code is running
    - E.g. a function variable’s scope is when the function is executing
  - Mostly used by domain-specific languages
    - E.g. bash, LaTeX, Emacs

Block Scope

- Name valid within the block its declared in
- Example: C++

```
if (a == 3) {
    int b = foo();    // b is valid inside this
    …                 // block only
}
std::cout << b;       // error: b not in scope
```

Function Scope

- Python is *different* from C++
- Variable declared in block available outside of block

```
def big(i):
    if i > 10:
          big = True     # boolean true in Python
          x = “so big”
    else:
          big = False    # boolean false
    print(big)           # this is valid, big in scope
    print(x)             # this is invalid if i <= 10
    return big           # function returns a boolean
```

Global Scope

- Inside function
  - Can access globals declared *anywhere*

```
def foo(a):
    print(CONST+a)
CONST = 5
foo(3)                    # prints 8
```

  - Declaring variable of same name *shadows* the global

```
CONST = 5
def foo(a):
    CONST = 6             # new local variable
    print(CONST+a)
foo(3)                    # prints 9
print(CONST)              # prints 5
```

Global Scope

- UnboundLocalError
  - Read global followed by *reassignment* of same name

```
CONST = 5
def foo(a):
    print(CONST+a)
    CONST = 6           # error: trying to write global
```

- Solution: *global* keyword

```
def foo(a):
    global CONST
    print(CONST+a)
    CONST = 6
foo(3)                  # prints 8
print(CONST)            # prints 6
```

Global Scope

- Global variable
  - Read and *update* are permitted without *global*

```
MUSIC = [ “Pop”, “EDM” ]
def retro():
    # empties the list and re-populate it
    MUSIC.clear()
    MUSIC.extend([ “Classic”, “Jazz” ])

>> retro()
>> print(MUSIC)
[‘Classic’, ‘Jazz’]
```

Default Argument

- Default value assigned to missing arguments
- In C++, default arguments always recreated

```
struct A {
    int x; A() : x(0) {}
    ~A() {
        cout << "destroyed\n";
    }
};

void foo(A a=A()) {
    cout << a.x << endl; a.x = 5;
}
```

```
int main() {
    foo();
    foo();
    return 0;
}
```

```
$ ./foo
0
destroyed
0
destroyed
```

Default Argument

- In Python, only evaluated *once*, when defined
  - Beware of mutable default arguments!

```
def add_topping(budget, toppings=list()):
    if budget > 4.99:
        budget -= 4.99
        toppings.append(“chipotle steak”)
    …
    return toppings

>> pizza1 = add_topping(5)
>> pizza2 = add_topping(4)
>> add_topping(3)
[‘chipotle steak’, ‘grilled chicken’, ‘broccoli’]
```

Default Argument

- Workaround
  - Convention: use None

```
def add_topping(budget, toppings=None):
    if toppings is None:
        toppings = list()
    …
    return toppings

>> pizza1 = add_topping(5)
>> pizza2 = add_topping(4)
>> add_topping(3)
[‘broccoli’]
```

Keyword Arguments

- Specify an argument using parameter name
- Useful for skipping over default arguments
- Also improves readability of arguments

```
def pizza(size=14, dough=“regular”, sauce=“tomato”, \
          cheese=“mozzarella”, toppings=()):
    …

# dough is regular
mine = pizza(16, sauce=“alfredo”, cheese=“cheddar”)

# all arguments before toppings use default values
yours = pizza(toppings=[“bacon”, “pepperoni”, “salami”])
```

Variadic Function

- Allows you to take variable number of arguments
  - Both positional and/or keyword arguments

```
def foo(*args, **kwargs):
    print(args, kwargs)

>> foo(1, 2, bar=3, baz=4)
((1, 2), {'baz': 4, 'bar': 3})

# only accepts one position, followed by keyword arguments
def foo(x, **kwargs):
    print(kwargs)

>> foo(1, 2)
TypeError: foo() takes 1 positional argument but 2 were given
```