COMP10001 Foundations of Computing
Advanced Functions

Semester 1, 2019
Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— VERSION: 1449, DATE: MARCH 1, 2019 —

Lecture Agenda

- This lecture:
  - Debugging and Testing (continued)
  - Functions and mutability
  - Parameters and arguments
  - Namespaces

Lecture Outline

1. Debugging (continued)
2. Testing
3. Functions and Mutability
4. Namespaces
5. Returning early
6. Parameters and arguments

Prevention Rather than Cure: Defensive Programming

- Build up your code bit by bit, using functions copiously, testing as you go against known inputs/outputs
- "Log" each step of your progress
- Use comments to remind you about any assumptions made by the code/corners cut along the way
- Always be on the lookout for common gotchas
- Above all, remember that the program code must communicate with humans, not just machines

Common Python Gotchas

- Equality (==) vs. assignment (=)
- Printing vs. returning from functions
- Correct use of types (e.g. `False` vs. `"False"`)
- Incorrect use of function/method (e.g. `return list.sort()`)
- Spelling and capitalisation
- Loops and incrementing
- Conditionals and indentation
- Namespace problems

A General Approach to Debugging

- Reproduce the bug
- Determine exactly what the problem is
- Eliminate "obvious" causes (e.g. *Is it plugged in?*)
- Divide the process, separating out the parts that work from the part(s) that don't ("isolate" the problem)
- When you reach a dead end, reassess your information; then step through the process again
- As you proceed, make predictions about what should happen and verify the outcome

Lecture Outline

The Actuality of Software Testing

- Execution-based verification:
  - generate and execute test cases, and check the correctness of the output
  - generally impossible to enumerate all possible inputs/use cases, so instead focus on developing a set of "representative" inputs to test the code over
  - in addition to "integration" testing (between system components), "unit" test the components of the system
- Non-execution-based verification
  - detect bugs by eyeballing the code directly, e.g. via code review or pair programming

Test Cases

- Test cases should be designed independently of the software implementation, and (ideally) be designed to:
  - test one thing each (e.g. one use case per input type)
  - test over the spectrum of use cases for the software (often based on the "boundaries" of inputs)
  - (in part) identify and test "corner case" inputs

What's with the Hidden Tests in Grok?

- For most real-world problems, the range of possible inputs to a function/system is infinite (or at least very, very large), making it impossible to test all possible inputs
- Instead, we construct a set of "hidden" (functional) tests to assess the generality of your code, by identifying different classes of input (including any "corner cases", or extrema), and manually constructing tests for each

What's with the Hidden Tests in Grok?

- For example, in the case of the following:

```
def substrn(sup, sub):
    '''
    Calculate the number of times `sub`
    occurs in `sup`
    '''
    sub_len = len(sub)
    n = 0
    for i in range(0, len(sup) - sub_len):
        if sub and sup[i:i+sub_len] == sub:
            n += 1
    return n
```


What's with the Hidden Tests in Grok?

- Corner cases = `sub` and/or `sup` are empty strings:

```
substrn('a', '') == 0
substrn('', '') == 0
```

- Class 1: `sub` and `sup` are the same string:

```
substrn('a', 'a') == 1
substrn('baba', 'baba') == 1
```

- Class 2: `sub` occurs multiple times in `sup`, possibly with overlaps between occurrences:

```
substrn('babababa', 'ba') == 4
substrn('aaa', 'aa') == 2
```

What's with the Hidden Tests in Grok?

- Class 3: sub does not occur in sup:

```
substrn('aaaa', 'bb') == 0
substrn('aabaa', 'aaa') == 0
substrn('aa', 'aaa') == 0
```


Bonus Python Tip: Automatically Running Tests

- As we are developing complex code, it makes sense to automatically test our code as we go. One way of doing this is via:

```
def myfun(...):
    ...

if __name__ == "__main__":
    print(myfun(...))
```

where `__name__` is set to `"__main__"` if and only if the program is "run" directly (as distinct from being imported etc.)

Lecture Outline

Local Variables and Mutability

- When you pass a mutable object to a function and locally mutate it in the function, the change is preserved in the global object:

```
def changeList(lst):
    lst = []
    return lst
def changeListItem(lst):
    lst[0] = "Changed, hah!"
```


Local Variables and Mutability

- As always Python Tutor is our friend:
  `http://www.pythontutor.com`

```
>>> mylist = [1,2,3]
>>> changeList(mylist)
[]
>>> mylist
[1, 2, 3]
>>> changeListItem(mylist)
>>> mylist
['Changed, hah!', 2, 3]
```


Local Variables and Mutability

- In fact, there is nothing specific to functions going on here; it is consistent with the behaviour of mutable objects user assignment/mutation:

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list2[0] = "Changed, hah!"
>>> list2
['Changed, hah!', 2, 3]
>>> list1
['Changed, hah!', 2, 3]
```

Lecture Outline

1. Debugging (continued)
2. Testing
3. Functions and Mutability
4. Namespaces
5. Returning early
6. Parameters and arguments

Namespaces

- A “namespace” is a mapping (dictionary!) from names to objects (e.g. variables and functions).
- When Python starts up there is the global namespace
- When a function is called, a local namespace for that function is called, and then forgotten when the function ends
- Scope is the area of Python code where a particular namespace is used

Namespaces

```
a = 3
def f(x):
    i = 2
    return x+i
b = 6
```

- In this code snippet
  - The global namespace contains `a`, `f` and `b`
  - When `f` is called, its local namespace has `x` and `i`
- When Python tries to find an object, it first looks in the local namespace, and then in the global namespace

Namespaces

```
1 i = 3
2 def f(x):
3     i = 1
4     return x+i
5 print(f(10))
```

- In this case, the Line 4 code uses the `i` in its local namespace (Line 3)
- Scope of `x` is Lines 2,3,4.
- Scope of global `i` is Lines 1, 2 and 5.
- Scope of the `i` in `f` is Lines 3 and 4.

Namespaces

- Now for the tricky part: functions within functions

```
i = 3
def f(x):
    def g(x):
        i = 1
        return x+i

    i = 2
    return g(x+i)

print(f(10))
print(g(10))
```

- `f`'s namespace contains `g` and others

Namespaces

```
def f(x):
    def g(x):
        return x+i

    i = 1
    return g(x+i)

print(f(10))
```

- Python searches local namespace, and then enclosing function namespaces, and then global namespace.
- You can list the current namespace with `dir()`.

Make Life Easy

- Don't use the same parameter names in sub-functions
- Avoid global variables wherever possible (always!)
- Capitalize constants (a common convention)

```
def f(x):
    ADDER_F = 2

    def g(y):
        ADDER_G = 1
        return(y + ADDER_G)

    return(g(x + ADDER_F))
```


Lecture Outline

Returning Early

If your function has the answer it needs, you can return straight away.

```
def any_fail(myList):
    """
    Returns True if any mark below 50,
    False otherwise. (Inefficient)
    """
    hasFail = False
    for mark in myList:
        if mark < 50:
            hasFail = True

    return hasFail
```


Returning Early

```
def any_fail(myList):
    """
    Returns True if any mark below 50,
    False otherwise. (Smart!)
    """
    for mark in myList:
        if mark < 50:
            return True   # why wait?

    return False
```


Lecture Outline

Parameters and Arguments

To allow us to talk precisely about functions, we define

- Parameters are the names that appear in a function definition
- Arguments are the values actually passed to a function when calling it

From `https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter`

Parameters and Arguments

```
def count_pos(tup):    # tup is the parameter
    """Count the positive elements in tup."""
    count = 0
    for i in tup:
        if i > 0:
            count += 1
    return count

print(count_pos((-1,2,3))) # (-1,2,3) is the
                           #          argument
```

(Aside: this is a very common pattern of looping; remember it as a template for your own coding.)

Default Arguments

- We have already seen that parameters can be given default arguments:

```
def seconds_in_year(days=365):
    return days*24*60*60
```

```
>>> seconds_in_year()
31536000
>>> seconds_in_year(366)
31622400
```

- But what is the scope of a default argument value?

Default Arguments

```
NUM_DAYS_IN_YEAR = 365

def seconds_in_year(days=NUM_DAYS_IN_YEAR):
    return days*24*60*60
```

```
>>> seconds_in_year()
31536000
>>> NUM_DAYS_IN_YEAR = 100
>>> seconds_in_year()
```

- The default values are evaluated *once* at the point of function definition in the *defining* scope.

Default Arguments

- This means you must be careful with mutable default arguments

```
def add_on_end(value, lst=[]):
    lst.append(value)
    return lst

print(add_on_end(1))
print(add_on_end(2))
print(add_on_end(3))
```

```
print(add_on_end(1, []))
print(add_on_end(2, []))
print(add_on_end(3, []))
```


Default Arguments

- If you want a mutable default (e.g. empty list) but not shared between calls

```
def add_on_end(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(add_on_end(1))
print(add_on_end(2))
print(add_on_end(3))
```

- `None` is a predefined constant in Python that has no value.

Default Arguments

- Where can you put default arguments in the function definition?

```
def add_on_end(lst=[], value):
    lst.append(value)
    return lst

print(add_on_end(1))
```

```
  File "program.py", line 1
    def add_on_end(lst=[], value):
                  ^
SyntaxError: non-default argument follows
             default argument
```

Keyword Arguments

- So far we have been using *positional* arguments: arguments are matched to their parameters by their position.

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return None

x = f(1, 2)
```

- But we can also match based on keywords (parameter names)

```
x = f(1, d=2)
```


Keyword Arguments

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return None

x0 = f()          # f() missing 'a'
x1 = f(a=1, 7)    # Default before non-default
x2 = f(1, a=2)    # f() multiple values for 'a'
x3 = f(b=8)       # what's 'b'?

x4 = f(c=8, a=2, d=9)   # all good
```


Keyword Arguments

```
def f(a, c=3, d=4):
    print("{0} {1} {2}".format(a,c,d))
    return None

x0 = f()
x1 = f(a=1, 7)
x2 = f(1, a=2)
x3 = f(b=8)
x4 = f(c=8, a=2, d=9)
```