COMP10001 Foundations of Computing
Advanced Functions (cont.)

Semester 1, 2019

Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— version: 1531, date: April 15, 2019 —

Lecture Agenda

- Last lecture:
  - Debugging and Testing
  - Functions and mutability
  - Parameters and arguments
  - Namespaces
- This lecture:
  - Returning early
  - Parameters and arguments
  - The call stack

Reminders

- Project 1 due this Thursday
- Mid-semester test viewing 12:30–1:30 this Wed 17/5, in Doug McDonell 10.05 (bring along your student card)

Lecture Outline

1. Returning early
2. Parameters and arguments
3. Tracing functions

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

1. Returning early
2. Parameters and arguments
3. Tracing functions

Parameters and Arguments

To allow us to talk precisely about functions:

- **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(f"{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("{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)
```

Keyword Arguments

```
def f(a, c=3, d=4):
    print("{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
```

Lecture Outline

1. Returning early
2. Parameters and arguments
3. Tracing functions

And Now for Something Completely Different ...

- Perform each of the following tasks, as commanded by your "programmer":
  - count from 1 to 10
  - spell *computing* backwards
  - hop on your left leg 10 times
  - recite the following lines from Shakespeare:
    The quality of mercy is not strain'd,
    It droppeth as the gentle rain from heaven
    Upon the place beneath
- Perform each task on demand, interrupting the current task when asked to perform the next task, and returning to it when other tasks are done

Tracing Functions: The Call Stack

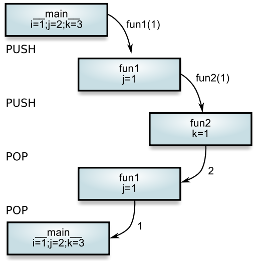
- We get some hints about how function "nesting" works from the Python interpreter:

```
def plus_one(i):
    return k + 1
print(plus_one(2))
```

```
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    print(plus_one(2))
  File "program.py", line 2, in plus_one
    return k + 1
NameError: name ’k’ is not defined
```

Tracing Functions: The Call Stack

- Functions are stored on the "call stack", facilitating function nesting, allowing functions to communicate with one another, and also preserving a function's local state/namespace



Tracing Functions: The Call Stack

- `http://pythontutor.com` shows the call stack

```
def a(x): print(x)
def b(x): return a(x)
def c(x): return b(x)
def d(x): return c(x)

d(10)
```

The Stack is Your Friend

- The stack trace in the message for run-time errors can often give you valuable hints on the cause of a bug:

```
1  def tofloat(i):
2      return flt(i)
3
4  def addnums(numlist):
5      total = 0
6      for i in numlist:
7          total += tofloat(i)
8      return total
9
10 nums = [1,2,3]
11 addnums(nums)
```

The Stack is Your Friend

```
Traceback (most recent call last):
  File "program.py", line 11, in <module>
    addnums(nums)
  File "program.py", line 7, in addnums
    total += tofloat(i)
  File "program.py", line 2, in tofloat
    return flt(i)
NameError: name 'flt' is not defined
```

From this, we can reproduce the sequence in which the functions were called, and *how* they were called, to be able to isolate the problem

The Stack is Your Friend

```
def to_int(x): return int(x)
def make_binary(x): return 'b' + x
def d(x): return to_int(make_binary(x))
print(d("101"))
```

```
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    print(d("101"))
  File "program.py", line 6, in d
    def d(x): return to_int(make_binary(x))
  File "program.py", line 2, in to_int
    return int(x)
ValueError: invalid literal for int() with
    base 10: 'b101'
```

Moral of the Story ...

```
Traceback (most recent call last):
  File "program.py", line 8, in <module>
    ...
  File "program.py", line 6, in f
   ...
WTFError:
```

- Python doesn't just print all that stuff for fun
- Make use of the stack trace to help you understand where your program went wrong

Lecture Summary

- Returning early from a function if we are done
- Be careful when passing mutable objects to functions
- What do we mean by “parameters” and “arguments”
- The scope of default arguments
- Using keyword arguments for additional flexibility
- The call stack is your friend