functions and modules

Ben Bolker

15:54 17 September 2019

Functions

Reference: Python tutorial section 4.6

- *the* most important tool for structuring programs
- allows *modularity*
- basic definition: `def function_name(args):` plus indented code block
- inputs are called **arguments**. outputs are called **return values**
- when function is called, go to the function, with the arguments, run code until you hit `return()` (return `None` if you get to the end without a `return`)

Return values

- most functions return values
- might not . . . *side effects*
  - input/output (create a plot, write output to a file, turn on a machine, . . .)
  - changing a (mutable!) variable

Function arguments

- basic arguments: *unnamed*, *mandatory*
- think of them as dummy variables; could be the same or different from the name in the calling environment

examples (try in Python tutor)

```
def add_one(x):
    x = x+1
    return(x)
x = 2
print("add_one=",add_one(x),", x=",x)
```

```
## add_one= 3 , x= 2
```

```
z = 2
print("add_one=",add_one(x),", x=",x)
```

```
## add_one= 3 , x= 2
```

`z` is **immutable** (a number), so it doesn't change; if you want it to change, use `z=add_one(z)`

mutability and functions

Changes within functions follow the standard mutability rules:

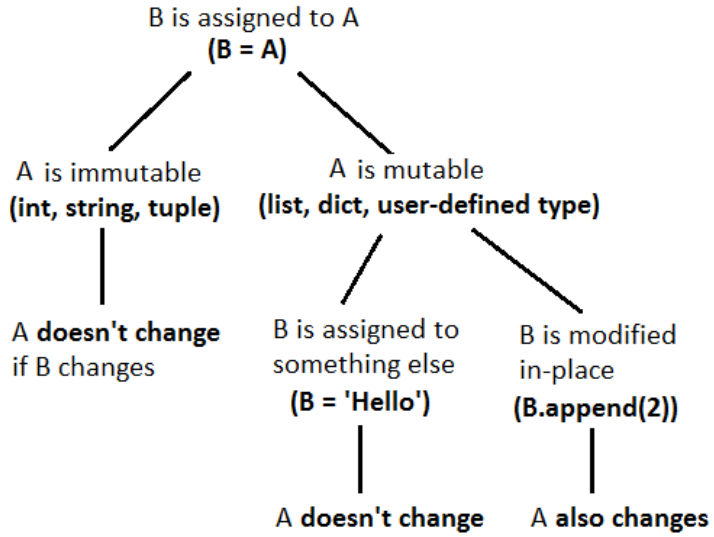

Figure 1: mutability mnemonic

Compare:

```
def no_return(x):
    x = [2,3,4]
    return(None)

z = [1,2,3]
no_return(z)
z

## [1, 2, 3]
```

With:

```
def no_return(x):
    x[0] = 7
    return(None)

z = [1,2,3]
no_return(z)
z

## [7, 2, 3]
```

optional arguments

- give *default* values
- for user convenience
- e.g. logarithm: `def log(value,math.e)`

Docstrings

- always say something about what your function does. (Feel free to give me a hard time in class if I don't.)

```
def documented_function():
    """this is a function that does
       nothing very useful
    """
    return(None)
```

Example

```
def add_function(a, b):
    """ the sum of two numbers
    Parameters
    ----------
    a : num
    b : num
    Returns
    -------
    sum : num
    The sum of a and b
    Examples
    --------
    >>> add_function(2, 5)
    7
    >>> add_function(3, -1.4)
    1.6
    """
    sum = a + b
    return sum
```

retrieving docstring

```
print(add_function.__doc__)
```

```
##  the sum of two numbers
##     Parameters
##     ----------
```

```
##     a : num
##     b : num
##     Returns
##     -------
##     sum : num
##     The sum of a and b
##     Examples
##     --------
##     >>> add_function(2, 5)
##     7
##     >>> add_function(3, -1.4)
##     1.6
##
```

Errors

Example code to work with

Types of errors

- **syntax errors** vs. **logic errors**
- a working matrix sum function
- failure modes from logic errors:
  - obvious failure
    - program stops with an error partway through: bad matrix sum #o
    - Python crashes
    - machine crashes
    - program never stops (infinite loop)
  - wrong answer
    - always vs. sometimes (obvious categories) vs. sometimes (mysterious)
    - obvious vs. subtle

Next section follows this presentation

- infinite loops:

What's wrong with this code? (It's meant to loop until the user enters either "y" or "n" . . .)

```
print("Please enter (y)es or (n)o")
cin = input()
while ((response != "y") or (response != "n")):
   print("Please try again")
```

or (`not response in "yn"`)
bad matrix #1

- operator precedence mistakes, e.g. $\Delta\text{fahrenheit} = \Delta\text{Celsius} \times 1.8$

```
fahrdiff = celsius_high - celsius_low * 1.8
```

- off-by-one error ("fencepost problem")
- ... more generally, **edge** or **corner cases**
- code incorrectly inside/outside loops:
- bad matrix #2
- bad matrix #3
- array index error (outside bounds)

Error messages

- error messages are *trying* to tell you something
- Google error messages (with quotation marks)

Debugging

- *brute-force logic* ("Feynman method"): stare at your code, try to figure out what's wrong
  (test cases: why is it failing in one specific situation?)
- flow charts, *pseudocode*
- tracing (`print()` statements)
  - put print statements before and after `if` conditions
  - before and after loops
  - in places where you suspect something might go wrong
- interactive tracing
- debugging tools (breakpoints/watchpoints/watches)

Searching for/asking for help

Searching for help

- Google (or your search engine of choice)
- be as specific as possible

Asking for help

- reproducible/minimal workable examples
  - right amount of context
  - "how to ask" (StackOverflow)
- browse/lurk in forums first!
- tone
- where:
  - forums
  - StackOverflow

Testing

- Simplify, simplify, simplify
- Reduce the size of your problem
- Cases with easy/known answers
- "corner" & "edge" cases
- Random tests (fuzz testing)
- Automatic testing framework: `nose`
  - built-in Python package
  - define test file
    - basic: `assert <condition>`
    - extra: `from nose.tools import assert_equal, assert_raises` (or something)
    - (generating an error: `raise ErrorType("message")`, e.g. `raise ValueError("non-conformable matrices")`
    - each test or set of tests as a separate function
    - see test_mm.py
  - `nosetests`/run in PyCharm
- Test-driven development: write tests **first**!

Additional resources

- `http://stackoverflow.com/questions/1623039/python-debugging-tips`
- `https://www.udacity.com/course/cs259`
- `http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29Debugging.html`
- `http://www.cs.cf.ac.uk/Dave/PERL/node149.html`