LECTURE 20

MORE ON FUNCTIONS, ARGUMENTS, AND ASSIGNMENT

MCS 260 Fall 2021

David Dumas

REMINDERS

- Project 2 due 6pm central today
- Project 2 solution will be posted next Friday
- Homework 7 posted, due 10am Tue

TERMINAL

I unified the "mini-terminal" examples from the 10am and 2pm lectures in `terminal.py`.

CRITICISMS

It's a good start, but:

- Adding a new command requires a new elif
- List of all commands (e.g. for `help`) must be manually updated

FUNCTIONS ARE VALUES

- Functions are values in Python, just like float, int, etc.
- Functions can be assigned to variables, used as parameters, stored in lists, used as keys or values

RETURNING MULTIPLE VALUES

```
def sumprod(x,y):
    """Return the sum and product of two numbers"""
    return x+y, x*y

s,p = sumprod(5,8)
# now s==13 and p==40
```

WHY THIS WORKS

A comma separated list (either bare or in parentheses) in Python is a **tuple**.

Tuples are like lists but immutable. They are iterable.

Tuple assignment lets you assign an iterable of values to a tuple of names as

```
name0, name1, name2 = value0, value1, value2
name0, name1, name2 = L  # if L has length 3
```

EXAMPLE: SWAP

```
x = 19
y = 52
x,y = y,x  # swap their values!
```

In other languages you would need a temporary place to store one of the values.

RETURNING MULTIPLE VALUES?

```
def sumprod(x,y):
    """Return the sum and product of two numbers"""
    return x+y, x*y

s,p = sumprod(5,8)
# now s==13 and p==40
```

From Python's perspective, `sumprod` returns one value (a tuple), and then tuple assignment stores those in `s` and `p`, respectively.

VARIADIC FUNCTIONS

A Python function can indicate that it will accept however many arguments the caller decides to give it:

```
def f(x,y,*args):
    """function that accepts 2 or more arguments"""
    # body of function here
    # probably examine len(args) and args[i], i=0,1,...
```

This example requires at least 2 arguments, but allows more. Arguments 3 and on are "packed" into a tuple called `args`.

ARGUMENT UNPACKING

Conversely, what if you know all the arguments you want to give a function, but they are in a list rather than separate variables?

```
L = ["Users","ddumas","teaching","mcs260","example.py"]
os.path.join(L) # FAILS
```

Use * to tell Python to **unpack** the list (or other iterable) into separate arguments:

```
L = ["Users","ddumas","teaching","mcs260","example.py"]
os.path.join(*L) # equivalent to os.path.join(L[0],L[1],...)
```

WRONG NUMBER OF ARGUMENTS

If you pass a function a number of arguments that it cannot accept, it raises **TypeError**. E.g.

```
def f(x,y,*args):
    print()

def g(x):
    print()

f()       # TypeError
f(1)      # TypeError
f(1,2)    # OK
f(1,2,3)  # OK

g()       # TypeError
g(1)      # OK
g(1,2)    # TypeError
```

BACK TO THE MINI-TERMINAL

Let's unify the many similar if/elif in our terminal example as follows:

- Make a dictionary to store all the commands
- Keys are command names
- Values are functions that perform the actions
- Main loop uses the command name to look up the right function to call. No if/elif/elif/...

REVISION HISTORY

- 2021-10-08 Initial publication