LECTURE 9

FUNCTIONS

MCS 260 Fall 2020

Emily Dumas

REMINDERS

- Work on:
  - Quiz 3 (due Today, 6pm central)
  - Project 1 (due Friday, 6pm central)
- Worksheet 4 available

We have seen lots of functions: `input()`, `print()`, `float()`, `len()`, `enumerate()`, ...

These are **built-in functions**, provided by Python. They do useful things, sometimes using data you provide, and sometimes returning a value.

It is also possible to create your own functions.

Syntax for a function definition:

```
def function_name(param0, param1, ...):
    statement
    ...
    statement
    return value
```

The **`param`*`i`*** are **parameters**.

Syntax for **calling** a function:

```
function_name(arg0, arg1, ...)
```

The **`arg`*`i`*** are **arguments**. The statements in the function body will run with **`param0=arg0`**, **`param1=arg1`**, ...

Function with no parameters

```
def input_yes_no():
    while True:
        s = input()     # Read string from keyboard
        s = s.lower()   # Make all lower case
        if s in ["y","yes"]:
            s = "yes"
            break
        elif s in ["n","no"]:
            s = "no"
            break
        else:
            print("Please enter y/yes or n/no.")
    return s
```

Now we can use this e.g. as:

```
print("Set all quiz scores to 100?")
if input_yes_no() == "yes":
    for i,student in enumerate(roster):
        scores[i] = 100.0
```

DOCSTRINGS

A Python function (or file) can begin with a string literal, a **docstring**, to document its purpose.

`help(function_name)` retrieves docstrings.

```
>>> def f(x):
...     """
...     Return the square of `x`.
...     """
...     return x*x
...
>>> help(f)
Help on function f in module __main__:

f(x)
    Return the square of `x`.
>>>
```

NEW RULE

Every function you write in MCS 260 must have a descriptive docstring.

A **return** is not required; a function can perform tasks without returning a value.

A return can appear anywhere in the function body to return to the caller immediately.

```
def input_yes_no2():
    """
    Read yes/no from keyboard, allowing single letter or full
    word answers.  Returns one of the strings "yes" or "no".
    """
    while True:
        s = input()     # Read string from keyboard
        s = s.lower()   # Make all lower case
        if s in ["y","yes"]:
            return "yes"
        elif s in ["n","no"]:
            return "no"
        else:
            print("Please enter y/yes or n/no.")
```

PARAMETERS

Parameters allow a function to accept and use data. The syntax is a list of names in parentheses after the function name. Example:

```
def trim(s, maxlen):
    """Return the initial segment of sequence s,
    consisting of at most `maxlen` items."""
    return s[:maxlen]  # Works even if s is short!
```

Now if we call `trim("picnic",3)`, the body of the function runs with `s="picnic"` and `maxlen=3`.

These are called **positional arguments**, as they correspond to parameters by position.

Parameters can be given default values:

```
def increase(x, addon=5):  # Note the default value for addon
    "Return the sum of `x` and `addon` (defaults to 5)"
    return x+addon
```

When calling a function, arguments can be given positionally, or by name. The latter are **keyword arguments**.

```
increase(3)             # result is 8
increase(3,addon=1)     # result is 4
increase(addon=2,x=3)   # result is 5
increase(addon=2,11)    # ERROR: pos. args must be first
increase(addon=2)       # ERROR: arg without default omitted
```

LOCAL VARIABLES

Variables and parameters changed inside a function don't affect anything outside of the function.

Such variables are **local**, and the function is their **scope**.

```
>>> def f():
...     "Example of local variables"
...     x = 10  # local variable
...     print("x is",x)
...
>>> x=3
>>> f()
x is 10
>>> x
3
```

REASONS TO USE FUNCTIONS

- **Don't repeat yourself** (DRY). Capture often-used code in a function to make programs smaller and easier to maintain.
- Well-named functions make the code using them more **readable**.
- Local variables provide **isolation**, avoid accidental modification or reuse of variables.

DRY

Consider

```
print("In celsius:")
print("Outside temp: ",(ext_air_f()-32)/1.8)
print("Inside temp: ",(int_air_f()-32)/1.8)
print("Forecast high (outside): ",(forecast_high()-32)/1.8))
```

versus

```
def to_celsius(fahrenheit_temp):
    "Convert Fahrenheit to celsius"
    return (fahrenheit_temp-32) / 1.8

print("In celsius:")
print("Outside temp: ",to_celsius(ext_air_f()))
print("Inside temp: ",to_celsius(int_air_f()))
print("Forecast high (outside): ",to_celsius(forecast_high()))
```

READABLE CODE

Short but dense:

```
for netid in [ x for x in roster if days_since_seen(x) > 7 ]:
    print("Not seen recently:",netid)
```

Longer but easier to understand:

```
def not_seen(netid,days=7):
    "Has this student been seen recently? Return bool"
    return days_since_seen(netid) > days

def students_not_seen(days=7):
    "List netids of students not seen in `days` days."
    return [ x for x in roster if not_seen(x,days=days) ]

for netid in students_not_seen():
    print("Not seen recently:",netid)
```

Often we care about *what* a function does, not *how*.

ISOLATION

Variable `t` used only briefly:

```
t = s.lower()
if t[0] == t[-1]:
    ...
```

Replace with local variable:

```
def first_and_last_same(x):
    """Does string `x` have same first and last
    letter (case insensitively)?"""
    x = x.lower()
    return x[0] == x[-1]

if first_and_last_same(s):
    ...
```

REFERENCES

- In *Downey*:
  - Chapter 3 and Chapter 6 both discuss functions, though the latter has a lot of material we didn't cover today (e.g. recursion)
  - Section 13.5 discusses keyword args

ACKNOWLEDGEMENT

- Some of today's lecture was based on teaching materials developed for MCS 260 by Jan Verschelde.

REVISION HISTORY

- 2020-09-14 Correction about keyword/positional arguments
- 2020-09-13 Initial publication