Lecture 8 - Defining functions

Computer Programming

Robert Vincent and Samia Hilal

Marianopolis College

March 9, 2022

What are functions?

- Programs involve many repeated operations.
- It is useful to *decompose* a large program into many *abstract* pieces.
- *Functions* (or *procedures*) are basic units of program construction.
- A function is *called* by another program or function.
- When a function completes, it *returns* a value to the *caller*.

Functions in Python

- Functions generally consist of:
  - a name.
  - a set of input parameters.
  - a return value.
- A function may also have *side effects*.
- Example: `print()`
  - Accepts any number of inputs.
  - No return value.
  - Side effect: prints to an output device.
- Example: `len()`
  - Accepts a single input parameter.
  - The return value is an integer, the length of the parameter.
  - No side effects.

The def statement

- We define a function using the syntax:

```
def name ( parameters ):
    statement1
    statement2
    ...
```

- The *parameters* are a comma-separated list of names for arguments passed in the function call.
- Both the function and parameter names must be legal Python names.
- The indented statement list forms the *body* of the function.

How the `def` statement works

- A `def` statement creates a new name that refers to the new function.
- The body of a function does not run until the function is *called*.
- Parameter names are private, so you can use any name, even those used elsewhere in your program.
- Variables created in a function are also private.
- We call these private names *local* variables.

The `return` statement

- A `return` statement defines the value of a function call.
- When a `return` statement runs, the program immediately exits the function body.
- The syntax is:

  `return` *expression*
- The expression can have *any* type.
- The value `None` is returned if either the expression is omitted or the function has no `return` statement.

The simplest possible function

```
def just_return_2():   # No parameters
    return 2           # Always returns 2

y = just_return_2()
print(y)               # Will print 2
```

- The function's name is `just_return_2`.
- It takes no arguments (empty parentheses).
- It returns a constant.

A slightly more useful case

```python
def increment(value):
    return value + 1  # Just add one.

x = increment(5)
print(x)            # Will print 6!
print(increment(x)) # Will print 7!
```

- The function's name is `increment`.
- It takes one argument (a number).
- It returns one plus its argument.

How to think about functions

- Parameters are the *inputs* to the function.
- The return value is the main *output* of the function.
- When called, each argument is matched to a parameter, usually by position.
- The value of the function call is whatever is returned.
- A function acts like a “black box”, we pass it some arguments and it computes its result. We can ignore the details.

Some other examples

```
def average(numbers):
    #Calculate the average for list of numbers
    return sum(numbers) / len(numbers)

def factorial(x):
   r, n = 1, 1
   while n <= x:
     r *= n
     n += 1
   return r

def printValue(x):
   print(’The number is’, x)
   # No return statement, so returns None
```

More about `return`

- A `return` is only legal in a `def` statement!
- A function can have *many* `return` statements.
- The first `return` encountered ends the function and defines its value.
- Example: l8mr.py

```
def factorial(x):
    if x < 0:
        print("Negative input.")
        return None # Signal an error
    r, n = 1, 1
    while n <= x:
        r *= n
        n += 1
    return r        # Return actual result.
```

Multiple return statements

- Example: l8mr2.py

```
def example(n, m):
    if n < m:
        return -1
    elif n > m:
        return 1
    else:
        return 0

print(example(1, 2)) # -1
print(example(2, 1)) # 1
print(example(2, 2)) # 0
```

Documenting functions

- A function may begin with a string (often triple-quoted).
- This *docstring* describes the function's purpose, arguments, and/or return value.
- Example: l8gcd.py

```
def gcd(a, b):
    '''Compute the greatest common divisor
       of positive integers 'a' and 'b'.'''
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```

Documenting functions

- Python stores this string with the function.
- The docstring is used by IDLE to provide the call tip.
- Also the Python `help()` function.
- Example: l8sqrt.py

```python
def square_root(x):             # Local x
    '''Return the square root of 'x'.

    Uses Newton's method to compute the
    square root to a fixed precision.'''
    y = x / 2                   # Local y
    while abs(y * y - x) > 1e-10:
        y = (y + x / y) / 2
    return y
```

Creating and using a function

- A function must be defined before calling it.
- When we call a function, we provide a list of arguments.
- The arguments are associated with the parameter names.
- The body of the function runs with those values.
- When the function returns, control returns to the caller.
- The value of the function call is the value given in the `return` statement that ended the function.

Local variables

- Any variable name can be used, or re-used, inside a function.
- Assigning a new value to a name creates a new, *local*, variable.
- This local variable exists **only** from the time it is created until the function completes (returns).
- Parameters and any variables created inside the function are *local* to the function. They exist for the duration of the function.
- They can have the same names as global names, or local variables in other functions.

Function parameters

- Any variable name can be used, or re-used, as a function parameter.
- The values of the arguments are assigned to the corresponding parameter names when the function is called.
- Every time the function is called, the parameter names are assigned to possibly different values.
- Normally the relationship between the arguments and parameters are determined by their order only.

Local variable example

- Example: l8local.py

```
def factorial(x):
    ’’’Compute the factorial of ’x’.’’’
    r, n = 1, 1
    while n <= x:
        r *= n
        n += 1
    return r

# The ’r’ and ’n’ below have no
# automatic relationship to those
# used in the function!!!
r = int(input(’Enter a number: ’))
n = factorial(r)
print(’The factorial of’, r, ’is’, n)
```

Local variable example 2

- Example: l8local2.py

```
def factorial(x):
    ’’’Compute the factorial of ’x’.’’’
    r, n = 1, 1
    while n <= x:
        r *= n
        n += 1
    return r

# Choosing different names here
# leaves us with the exact same
# program, functionally!!
x = int(input(’Enter a number: ’))
y = factorial(x)
print(’The factorial of’, x, ’is’, y)
```

Variable scopes 1

- A variable name created within a function normally exists only as long as the function is executing.
- The rules that govern the lifetime of names in a language are called *scope* or *scoping* rules.

```
>>> def add(a, b):
...     result = a + b # A local variable!
...     return result
...
>>> print(add(10, 7))
17
>>> print(result) # Not defined globally.
NameError: name 'result' is not defined
```

Variable scopes 2

- Variables created inside a function are often called *local* variables.
- Local variables and parameters can share names with *global* variables elsewhere in the program.

```
>>> def add(a, b):
...     result = a + b # Local to ’add’
...     return result
...
>>> a, result = 8, 7 # Globals
>>> print(add(10, a))
18
>>> print(result, a)
7 8
```

Changing function parameters 1

- For immutable types, changing parameters only affects the function body:

```
>>> def factorial(x):
...   r, n = 1, 1
...   while n <= x:
...     r *= n
...     n += 1
...   return r
...
>>> r = 10
>>> print(factorial(r))
3628800
>>> print(r)
10
```

Changing function parameters 2

- Again, most changes will be invisible to the calling program:

```
>>> def swap(x,y):
...   print(’before’,x,y)
...   x,y = y,x
...   print(’after’,x,y)
...
>>> m,n = 5,10
>>> swap(m,n)
before 5 10
after 10 5
>>> print(m,n)
5 10
```

Changing function parameters 3

- Mutable parameters *may* be changed within a function:

```
>>> def prepend(num_list, value):
...   num_list.insert(0, value)
...
>>> x = [1,2,3]
>>> prepend(x, 5)
>>> print(x)
[5, 1, 2, 3]
>>> prepend(x, 1)
>>> print(x)
[1, 5, 1, 2, 3]
```

Changing function parameters 4

- Mutable values can be changed, but you can't change which variable references a particular list:

```
>>> def swap(x,y):
...     x,y=y,x
...
>>> a,b = [1,2,3],[4,5,6]
>>> swap(a, b)
>>> print(a)
[1, 2, 3]
>>> print(b)
[4, 5, 6]
```

Checking argument lists 1

- Python is *not* picky about the *type* of arguments you pass to a function.
- A function may therefore support multiple types:

```
>>> def add(a, b):
...     return a + b
...
>>> print(add(3.1, 6.8)) # float
9.9
>>> print(add(’Py’, ’thon’)) # str
Python
>>> print(add((1, 2), (3, 4))) # tuple
(1, 2, 3, 4)
```

Checking argument lists 2

- Python *is* picky about the *number* of arguments you pass to a function.

```
>>> def add(a, b):
...     return a + b
...
>>> print(add(4))
TypeError: add() missing 1 required
positional argument: 'b'
>>> print(add(1, 2, 3))
TypeError: add() takes 2 positional arguments
but 3 were given
>>>
```

Introduction to recursion

- *Recursion* is a common concept in mathematics and computer science.
- A function is *recursive* if it is defined, in part, in terms of itself.
- This may sound like a problem, or an error, but in fact it is often quite useful.
- It means that a function may call *itself* in at least some cases.

Recursion in practice

- Recursion works because the function is not *called* when it is *defined*.
- A correct recursive definition must include:
  - A simple *base case* that terminates the recursion.
  - A formal procedure that reduces all other cases to the base case.
- Most recursive functions call *themselves* directly.
- Others may be *mutually recursive*, e.g. `f()` calls `g()`, and `g()` calls `f()`.

Recursion: factorial

- The factorial function is a classic example of a recursive function:
  - Base case: $0! = 1$
  - Recursive case: $N! = N \times (N-1)!$
  - Example: l8fact_r.py

```
def fact(n):
    '''Recursive factorial'''
    if n <= 0:
        return 1
    else:
        return n * fact(n - 1)

print(fact(10)) # 3628800
```

Recursion: Fibonacci numbers

- Base case: $f(0) = 0$, $f(1) = 1$
- Recursive case: $f(n) = f(n-1) + f(n-2)$
- Example: l8fib_r.py

```
def fib(n):
    if n <= 1:
        return n
    else:
        # 2 recursive calls.
        return fib(n - 1) + fib(n - 2)

for i in range(10):
    print(fib(i), end=’ ’)
print()
```

Recursion: Sum of a list

- Base case: `sum_r([])` is 0
- Recursive case: `lst[0] + sum_r(lst[1:])`
- Example: l8sum_r.py

```python
def sum_r(lst):
    '''Recursive sum of a list.'''
    if len(lst) != 0:
        return lst[0] + sum_r(lst[1:])
    else:
        return 0

print(sum_r([10, 5]))          # 15
print(sum_r([1, 5, 9, 6, 11])) # 32
print(sum_r([]))               # 0
```

Recursion: Reverse a list

- Base case: `rev_r([])` is `[]`
- Recursive case:

  `rev_r(x) = rev_r(x[1:]) + x[:1]`
- Example: l8ev_r.py

```
def rev_r(lst):
    if len(lst) > 0:
        return rev_r(lst[1:]) + lst[:1]
    return []

x = [9, 7, 4, 1, 5]
print(rev_r(x)) # [5, 1, 4, 7, 9]
```

Advantages of recursion

- Some problems have an inherently recursive structure:
    - Languages (human and programming)
    - Games
    - Many math problems (e.g. factorial)
    - etc.
- For these problems, recursion can be simpler than iteration.
- It is usually possibly define a function using either iteration or recursion.

Problems with recursion

- Iteration is often easier to understand.
- Most recursive functions can be defined as iterative as well.
- Iteration, when practical, is usually cheaper in terms of both memory and time.

Summary

- Functions are a convenient way to decompose a program into smaller units.
- We define a function with the `def` statement.
- A `return` statement ends the function call and gives it a value.
- Functions must be defined before we use them.
- Function parameters and local variables are private to the function.
- Recursion is a programming technique in which a function is defined in terms of *itself*.