Australian National University

COMP1730/COMP6730
Programming for Scientists

Functions (part 2)

Lecture outline

* Recap of functions.
* Namespaces & references.
* Recursion revisted.

Functions (recap)

* A *function* is a piece of code that can be *called* by its name.
* Why use functions?
  – **Abstraction**: To use a function, we only need to know *what* it does, *not how*.
  – Readability.
  – Divide and conquer – break a complex problem into simpler problems.
  – A function is a logical unit of testing.
  – Reuse: Write once, use many times (and by many).

Function definition

name / parameters

```
def change_in_percent(old, new):
    diff = new - old
    return (diff / old) * 100
```

4 spaces

suite

* The function suite is defined by indentation.
* Function *parameters* are variables local to the function suite; their values are set when the function is called.
* The `def` statement only *defines* the function – it does not execute the function.

Function call

- To call a function, write its name followed by its *arguments* in parentheses:

  ```
  change_in_percent(364, 489)
  ```

- Order of evaluation: The argument expressions are evaluated left-to-right, and their values are assigned to the parameters; then the function suite is executed.
- `return` *`expression`* causes the function call to end, and return the value of the expression.

Functions without return

* A function call is an expression: its value is the value `return`'d by the function.
* In python, functions always return a value: If execution reaches the end of a function suite without executing a `return` statement, the return value is the special value `None` of type `NoneType`.
* Note: `None`-values are not printed in the interactive shell (unless explicitly with `print`).

Namespaces

Namespaces

* Assignment associates a (variable) name with a reference to a value.
  - This association is stored in a *namespace* (sometimes also called a “*frame*”).
* Whenever a function is called, a new *local namespace* is created.
* Assignments to variables (including parameters) during execution of the function are done in the local namespace.
* The local namespace disappars when the function call ends.

Scope

* The *scope* of a variable is “the set of program statements over which a variable exists (i.e., can be referred to)”.
  – In other words, the set of program statements over which the namespace that the variable is defined in persists.
* Because there are several namespaces, there can be *different variables with the same name in different scopes*.

```
def f(x):
    y = x ** 2
    return y - 1
x = 1
y = f(x + 1)
```

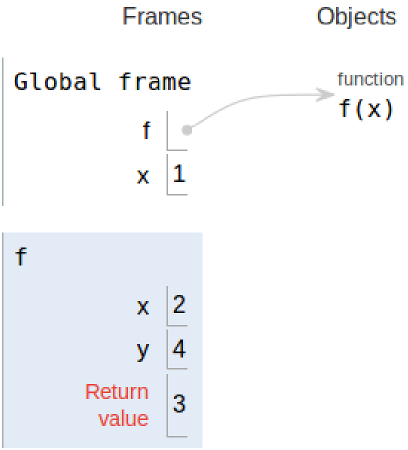


Image from `pythontutor.com`

```
def f(x):
    y = x ** 2
    return y - 1

x = 1
y = f(x + 1)
```

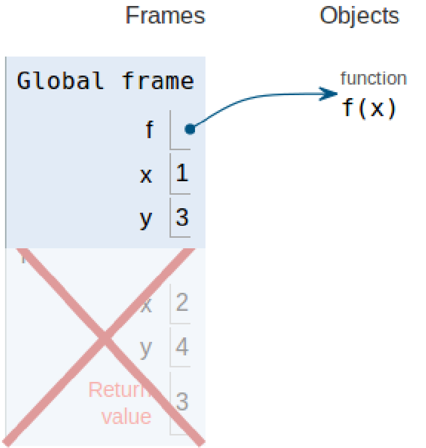


Image based on pythontutor.com

The local assignment rule

* python considers a variable that is assigned **anywhere** in the function suite to be a "*local variable*" (this includes parameters).
* When a non-local variable is evaluated, its value is taken from the (enclosing) global namespace.
* When a local variable is evaluated, only the local namespace is checked.
  – If the variable is not defined there, python raises an `UnboundLocalError`.
* The rule considers only *variable assignment*.

```
def f(x):
    return x ** y
>>> y = 2
>>> f(2)
4
```

```
def f(x):
    if y < 1:
        y = 1
    return x ** y
>>> y = 2
>>> f(2)
UnboundLocalError:
 local variable 'y'
 referenced before
 assignment
```

* Modifying is not assignment!
  - Assignment changes/creates the association between a name and a reference (in the current namespace).
  - A modifying operation on a mutable object – including index and slice assignment – does not change any name–value association.

```
def f(x):
    y = x ** 2
    f_list.append([x,y])
    return y
>>> f_list = []
>>> f(2)
4
>>> f(3)
9
>>> f_list
[[2, 4], [3, 9]]
```

Argument values are references

* When a function is called, its parameters are assigned *references* to the argument values.
  - If an argument value refers to a mutable object (for example, a list), modifications to this object made in the function are visible outside the function’s scope.

```
def f(ns):
    total = 0
    while len(ns) > 0:
        next = ns.pop(0)
        total = total + next
    return total
```

```
>>> a_list = [1,2,3]
>>> f(a_list)
6
>>> a_list
[]
```

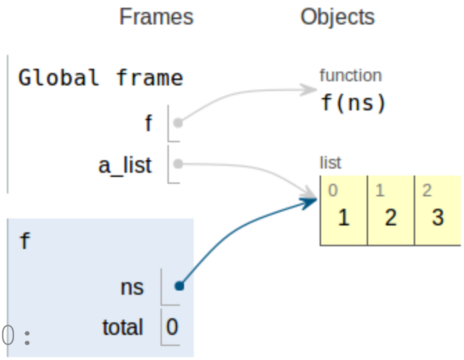


Image from pythontutor.com

```
def f(ns):
    total = 0
    while len(ns) > 0:
        next = ns.pop(0)
        total = total + next
    return total

>>> a_list = [1,2,3]
>>> l_sum = f(a_list)
```

Other namespaces

* python’s built-in functions are defined in a separate namespace; it is always searched last if a name is not found elsewhere.
* Imported modules are executed in their own namespace.
  – Names in a module namespace are accessed by prefixing the name of the module.
* User-defined classes and objects (not covered in this course) also have their own namespace

Guidelines for good functions

- Within a function, *access only local variables*.
  - Use parameters for all inputs to the function.
  - Return all function outputs (for multiple outputs, return a tuple or list).
  - ...except if the *specific purpose* of the function is to send output elsewhere (e.g., print).
- Don’t modify mutable argument values, unless the *specific purpose* of the function is to do that.
- **Rule #4**: No rule should be followed off a cliff.

Australian National University

Recursion

* A recursive function is often described as "a function that calls itself".
* Function calls form a *stack*: when the *i*th function call ends, execution returns to where the call was made in the $(i - 1)$th function suite.
* The function suite must have a branching statement, such that a recursive call does not always take place ("base case"); otherwise, recursion never ends.
* Recursion is a way to think about how to solve problems: reducing it to a smaller instance of itself.

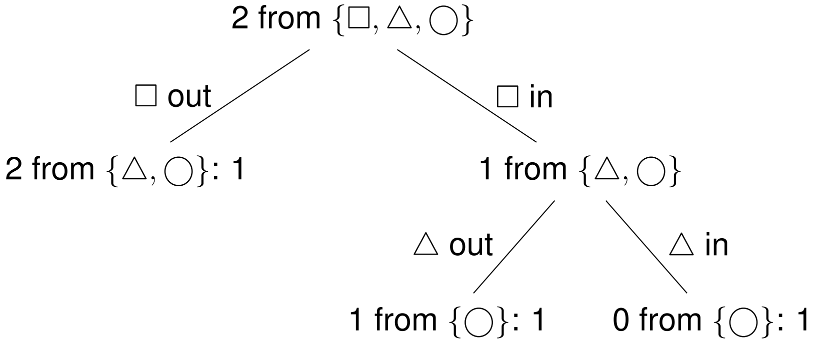
Example (contrived)

```
def f(x):
    '''Returns 2 ** x.
    x is an integer >= 0.
    '''
    if x == 0:
        return 1        # base case
    else:
        y = f(x - 1)    # recursive call
        return 2 * y
```

```
1 def f(x):
      ...
2 y = f(2)
            x = 2
          3 if x == 0:
          4 else:
          5 y = f(x - 1)
                          x = 1
                        6 if x == 0:
                        7 else:
                        8 y = f(x - 1)
                                        x = 0
                                      9 if x == 0:
                                     10 return 1
                          x = 1, y = 1
                       11 return 2 * y
            x = 2, y = 2
         12 return 2 * y
  y = 4
```

Example: Counting selections

- Compute the number of ways to choose a subset of k elements from a set of n, $C(n, k)$.



* Recursive formulation:

$$
\begin{aligned}
C(n,k) &= C(n-1,k) + C(n-1,k-1) \\
C(n,0) &= 1 \\
C(n,n) &= 1
\end{aligned}
$$

```
def choices(n, k):
    if k == n or k == 0:
        return 1
    else:
        return choices(n - 1, k) + \
               choices(n - 1, k - 1)
```

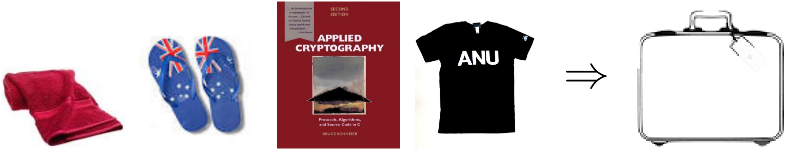
```
1 ans = choices(3,2)
      | n = 3, k = 2
      | 2 if k == 0 or k == n:
      | 3 else:
      | 4 choices(n - 1, k)
      |       | n = 2, k = 2
      |       | 5 if k == 0 or k == n:
      |       | 6 return 1
      | 7 choices(n - 1, k - 1)
      |       | n = 2, k = 1
      |       | 8 if k == 0 or k == n:
      |       | 9 else:
      |       | 10 choices(n - 1, k)
      |       |       | n = 1, k = 1
      |       |       | 11 if k == 0 or k == n:
      |       |       | 12 return 1
      |       | 13 choices(n - 1, k - 1)
      |       |       | n = 1, k = 0
      |       |       | 14 if k == 0 or k == n:
      |       |       | 15 return 1
```

```
 4 choices(n - 1, k)
             n = 2, k = 2
           5 if k == 0 or k == n:
           6 return 1
 7 choices(n - 1, k - 1)
             n = 2, k = 1
           8 if k == 0 or k == n:
           9 else:
          10 choices(n - 1, k)
                         n = 1, k = 1
                      11 if k == 0 or k == n:
                      12 return 1
          13 choices(n - 1, k - 1)
                         n = 1, k = 0
                      14 if k == 0 or k == n:
                      15 return 1
          16 return 1 + 1
17 return 1 + 2
ans = 3
```

Example: Subset sum

* Given a list of n integers $w_0, \ldots, w_{n-1}$, is there a subset of them that sums to exactly C?

  (Also known as the "(exact) knapsack problem":



$w_0 = 5$ $w_1 = 2$ $w_2 = 9$ $w_3 = 1$ $C = 16$.)

Example: Sudoku