COMP1730/COMP6730
Programming for Scientists

Functions, part 3

Announcements

* Major assignment due on Sunday - 11:55pm.
* Practice Examination Available
* Feedback and practice oral examinations will be available in the labs next week.
* Oral examination sign-up will go in Wattle next week.
* Next two Mondays are public holidays
* You can attend any lab next week.

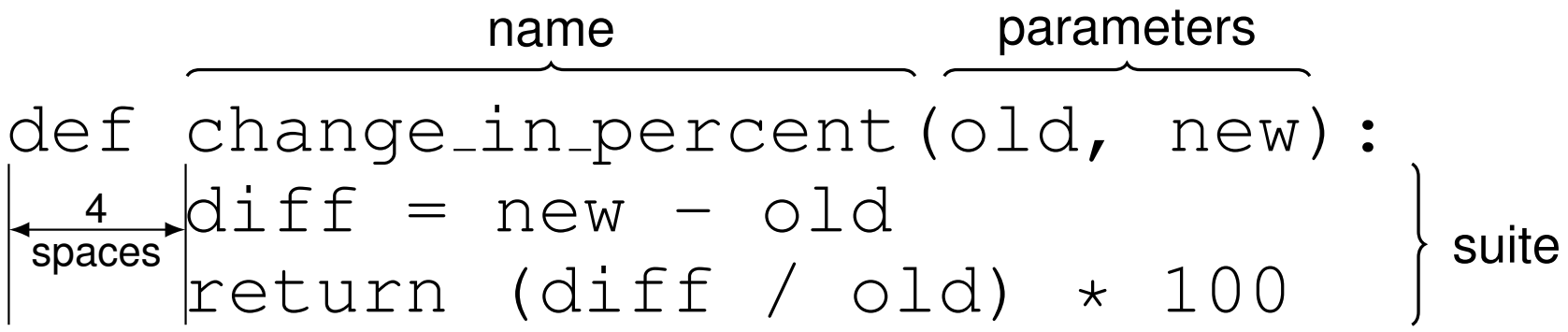
Lecture outline

* Recap of functions.
* Keyword arguments and parameter defaults.
* The function type in Python.
* Recursion

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



* 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(485, 523)
```

* 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.

Positional and keyword arguments

* By default, function call arguments are mapped to parameters by *position* (left-to-right).
* python also allows *named* (a.k.a. *keyword*) arguments (and a mix of both).

```
def log(x, b):
    ...

>>> log(3, 2)      # x = 3, b = 2
>>> log(3, b=2)    # x = 3, b = 2
>>> log(b=2, x=3)  # x = 3, b = 2
```

Parameter default values

* python allows function definitions to specify parameter default values.
* Parameters without a default value are *required*, and must precede all parameters with defaults.

```
def log(x, b = 2):
    ...
>>> log(3)            # x = 3, b = 2
>>> log(3, 10)        # x = 3, b = 10
>>> log(b=3, x=3)     # x = 3, b = 3
```

Why Use default and keyword parameters?

* Allows you to change a function signature without breaking existing code.
* Allows you to have more complex function signatures without making the user specify lots of parameters. For example `print`, `open` and many `matplotlib` visualisation functions.
* Don’t go overboard - too many parameters is (usually) a sign that you are trying to do too many different things.

Mutable objects as defaults

* Generally speaking not a good idea.

```
def a_func(a, b = []):
    b.append(a)
    return sum(b)

x = a_func(11)
y = a_func(12, [1, 3, 5])
z = a_func(13)
print(z)
>>> ?
```

The function type

Function definition

* A function definition is a variable assignment. The variable name is the function name and the value is an object of type `function`.
* For example:

```
def log(x, b):
    ...
```

* assigns an object of type `function` to the variable named *log*.

```
def log(x, b):
    '''A log function'''
    ...
>>> type(log)
>>> function
>>> ...
>>> log.__doc__
>>> 'A log function'
```

* You can do anything with this object you can do with any other type in python, for example:
  - reassign it

    ```
    log = 15
    ```
  - store it in a container

    ```
    my_list[0] = log
    ```

    or

    ```
    my_dict['log function'] = log
    ```
  - pass it as a parameter to another function

    ```
    func_2(a, b, log_function = log)
    ```

* Except reassignment, none of these actions stop you from calling the function.

```
def log(x, b):
    ...


my_dict['log function'] = log
...
my_dict['log function'](15, 3)
```

* Can be used to make your code more general, e.g. a function that solves an equation.

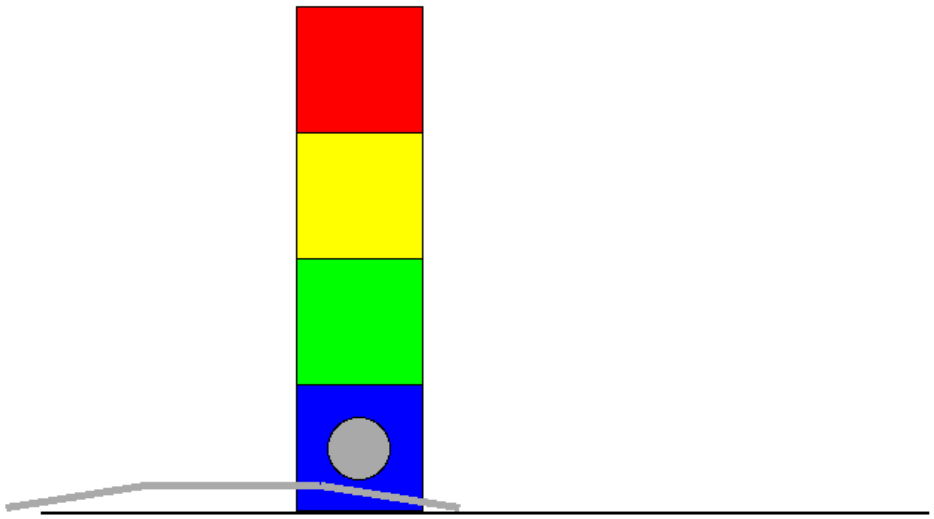
Recursion

Recursion

* The suite of a function can contain function calls, including *calls to the same function*.
  - This is known as *recursion*.
* 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 solving a problem: how to reduce it to a simpler instance of itself?

Problem: Counting boxes

* How many boxes are in the stack from the box in front of the sensor and up?

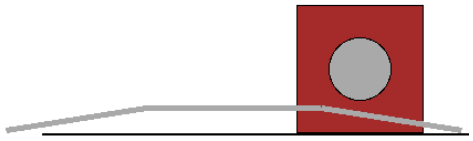


* If `robot.sense_color() == ''`, then the answer is zero.
* Else, one plus what the answer would be if the lift was one level up.

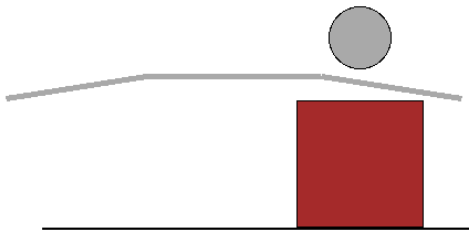
```
def count_boxes():
    if robot.sense_color() == '':
        return 0
    else:
        robot.lift_up()
        num_above = count_boxes()
        robot.lift_down()
        return 1 + num_above
```

The call stack (reminder)

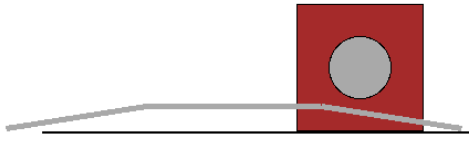
* When a function call begins, the current instruction sequence is put “on hold” while the function suite is executed.
* Execution of a function suite ends when it encounters a `return` statement, or reaches the end of the suite.
* The interpreter then returns to the next instruction after where the function was called.
* The *call stack* keeps track of where to come back to after each current function call.



```
1  ans = count_boxes()
2  if robot.sense_color() == '':
3      robot.lift_up()
```



```
4      num_above = count_boxes()
5      if robot.sense_color() == '':
6          return 0
7      num_above = 0
8      robot.lift_down()
```



```
9      return num_above + 1
10 ans = 1
```

Problem: Fibonacci numbers

* The Fibonacci numbers are the sequence: 0, 1, 1, 2, 3, 5, 8, 13, ...
* Mathematically we can define it as:
  - $F_n = 0$ if n = 1
  - $F_n = 1$ if n = 2
  - $F_n = F_{n-1} + F_{n-2}$ if $n > 2$
* What is F_{10}?