COMP1730/COMP6730
Programming for Scientists

Functions

Announcements

- Late enrolees: **Read the news forum** on wattle – including the announcements from last week.
- Homework 1 is due **11:55pm Sunday the 10th**.
- Homework 1 will be assessed in lab next week – you must attend your lab session.

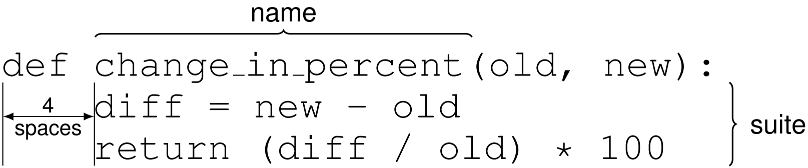
Lecture outline

- Function definition.
- Function calls & order of evaluation.
- Assignments in functions; local variables.
- Function testing & documentation.

Functions

- In programming, a *function* is a piece of the program that is given a name, and can be *called* by that name.
- Functions definitions promote *abstraction* (“what, not how”) and help break a complex problem into smaller parts.
- To encapsulate computations on data, functions have *parameters* and a *return value*.

Function definition (reminder)



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

- A function definition consists of a name and suite.
- The extent of the suite is defined by indentation, which must be the same for all statements in the suite (standard is 4 spaces).

Function definition

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

- Function *parameters* are (variable) names; they can be used (only) in the function suite.
- Parameters’ values will be set only when the function is called.
- `return` is a statement: when executed, it causes the function call to end, and return the value of the expression.

Function call

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

```
>>> change_in_percent(315, 262)
20.229007633587788
```

- The arguments are expressions.
- Their number should match the parameters.
  - Some exceptions; more about this later.
- A function call is an expression: its value is the value returned by the function.

Order of evaluation

- The python interpreter always executes instructions one at a time in sequence; this includes expression evaluation.
- To evaluate a function call, the interpreter:
  - First, evaluates the argument expressions, one at a time, from left to right.
  - Then, executes the function suite with its parameters assigned the values returned by the argument expressions.
- Same with operators: first arguments (left to right), then the operation.

The call stack

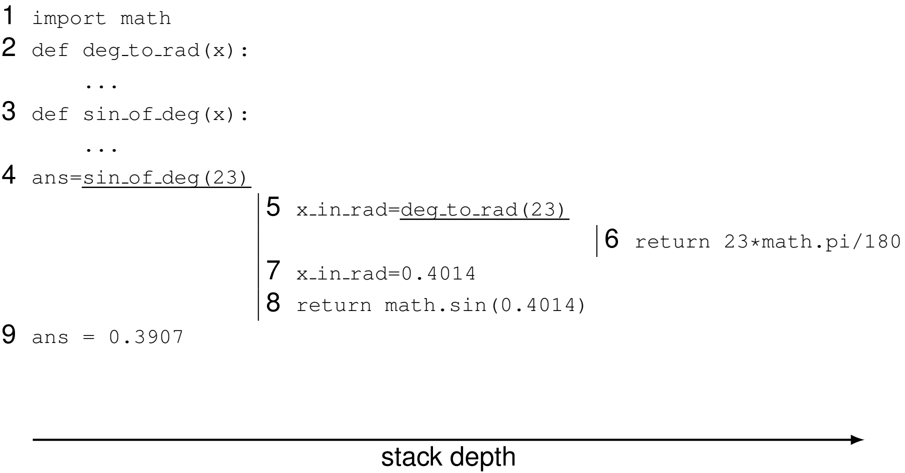
- When evaluation of a function call begins, the current instruction sequence is put “on hold” while the expression is evaluated.
- When execution of the function suite ends, the interpreter returns to the next instruction after where the function was called.
- The “to-do list” i.e. where to come back to after each function call ends, is called the *stack*.

```
import math

# Convert degrees to radians.
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees.
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
```



```
1 import math
2 def deg_to_rad(x):
      ...
3 def sin_of_deg(x):
      ...
4 ans=sin_of_deg(23)
                    | 5 x_in_rad=deg_to_rad(23)
                    |                         | 6 return 23*math.pi/180
                    | 7 x_in_rad=0.4014
                    | 8 return math.sin(0.4014)
9 ans = 0.3907
```

Assignments in functions

- Variables assigned in a function (including parameters) are *local* to the function.
  - Local variables are “separate” – the interpreter uses a new namespace for each function call.
  - Local variables that are not parameters are undefined before the first assignment in the function suite.
  - Variables with the same name used outside the function are unchanged after the call.
- The full story is a little more complicated – we’ll return to it later in the course.

Functions with no `return`

- If execution of a function suite reaches the end of the suite without encountering a `return` statement, the function call returns the special value `None`.
  - `None` is used to indicate “no value”.
  - The type of `None` is `NoneType` (different from any other value).
- In interactive mode, the interpreter does *not* print the return value of an expression when the value is `None`.

Side effects and return values

- An expression *evaluates to* a value.
- A statement does not return a value, but executing it causes something to happen, e.g.,
  - `a_number = 2 + 3` : variable `a_number` becomes associated with the value `5`;
  - `print(2 + 3)` : the value `5` is printed.

  This is called a *side effect*.
- We can write functions with or without side effects, and functions that do or don’t return a value (other than `None`).

- Functions with side effects and no return value:
  - `robot.drive_right()`
  - `print(...)`
- Functions with return value and no side effect:
  - `math.sin(x)`
  - `change_in_percent(old, new)`
- Functions with side effects and return value?
  - Possible.
- Functions with no side effect and no return value?

Functions of functions

- In python, functions are also values:

```
>>> type(change_in_percent)
...
```

- A function can be passed as an argument to another function:

```
def gradient(f, x, d):
    return (f(x + d) - f(x - d)) / (2*d)
ans = gradient(math.sin, math.pi/4, 0.1)
```

Testing and Documentation

Function testing

- A function is a logical unit of testing.
  - Specify the assumptions (for example, type and range of argument values);
  - Test a variety of cases under the assumptions.
- What are “edge cases”?
  - Typical (numeric) examples: values equal to/less than/greater than zero; very large and very small values; values of equal and opposite signs; etc.
- Remember that floating-point numbers have limited precision; == can fail.

```
>>> change_in_percent(1, 2)
100.0
>>> change_in_percent(2, 1)
-50.0
>>> change_in_percent(1, 1)
0.0
>>> change_in_percent(1, -1)
-200.0
>>> change_in_percent(0, 1)
ZeroDivisionError
```

The function docstring

```
def change_in_percent(old, new):
    ’’’Return change from old to new, as
    a percentage of the old value.
    old value must be non-zero.’’’
    return ((new - old) / old) * 100
```

- A *docstring* is a string literal written as the first statement inside a function’s suite.
- Acts like a comment, but accessible through the built-in help system.
- Describe *what* the function does (if not obvious from its name), and its *limits* and *assumptions*.