# COMP1730/COMP6730

## Programming for Scientists

# Functions

# 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

Python:

```
def change_in_percent(old, new):
    # function suite
    diff = new – old
    return (diff / old) ∗ 100
```

C/Java:

```
double change_in_percent(double old, double new) {
    double diff = new – old;
    return (diff / old) ∗ 100;
}
```

- A function definition consists of a name (`change_in_percent`) 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 *parameters* are (variable) names: `old`, `new`; 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(489, 556)
13.701431492842536
```

- The arguments are expressions.
- Their number should match the parameters.
- A function call is an expression: it's value is the value returned by the function.

# Programming problem: Rewrite neuron activity using function

```
# Example to describe activity of a neuron
# in a neural network
import math

# input signals
x1 = 0.7
x2 = 0.43

# weights of arrows
w1 = 3.2
w2 = 1.5

# bias to modify output independent of inputs
bias = –10

summation = w1∗x1 + w2∗x2 + bias
output = 1/(1+math.exp(–summation))

print(summation, " ", output)
```

# Function Call Execution

# 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, evalutes 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” of where to come back to after each current function call is called the *stack*.

Demonstrate the code in `https://pythontutor.com`

# 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.
- Back to pythontutor

# 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 `None` return:
  - `stanfordkarel.move()`
  - `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 `None` return value?

# Functions of functions

- In python, functions are also values: a function can be passed as argument to another function.

```python
import math
# weights of arrows
w1 = 3.2
w2 = 1.5
# bias to modify output independent of inputs
bias = -10

def sigmoid(x):
    return 1/(1+math.exp(-x))

def neuron_output(x1, x2, activation):
    total = w1*x1 + w2*x2 + bias
    return activation(total)

print(neuron_output(0.7, 0.43, sigmoid))
```

# Testing and Documentation

# Test, test, test

- How do we know our program works?
- A function is a logical unit of testing:
  - Specify the assumptions under which the function is meant to work, e.g., type and range of argument values
- Test a variety of cases under the assumptions, esp. “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
```

# Some common errors

- `SyntaxError`:
  You have broken the rules of python syntax
- `NameError` or `AttributeError`:
  You have used a (function) name that doesn't exist. Check for typos.
- `IndentationError`:
  Too much or too little indentation
  - All statements in a function suite must have the same indentation
  - All statements outside function definitions must have no indentation

# Example errors with Karel the robot

```
Traceback (most recent call last):
File "karel_test.py", line 8, in main
    fill_hole()
File "karel_test.py", line 4, in fill_hole
    pick_beeper()
KarelException: Karel crashed while on
                avenue 2 and street 1, facing East
Invalid action: Karel attempted to pick up a beeper,
                but there were none on the current
                corner
```

- Errors will happen when running your program
- Read the error message!

# 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*.