The UK's European university

University of Kent

COMP8270 / PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Fernando Otero
febo@kent.ac.uk
cs.kent.ac.uk/people/staff/febo

overview:

1. Functions
2. Arguments
3. Return Values

Python Functions

Why Use Functions?

- Re-use: write the code once, invoke from wherever is convenient.
- Scope: names and state inside a function are private
  - Easier to debug as change of state is isolated.
- Divide and conquer: decompose a problem in to smaller, discrete and tractable pieces.
- Recursion: an important special case of Divide/Conquer. Complicated problems and problems with recurrent state are easily implemented.
- Testing:
  - Another benefit of Divide/Conquer, test individual pieces in isolation (“unit testing”).
  - Higher probability assembled system is correct if composed of well-behaved components.

Functions: What Are They?

- *f: Domain → Range*
- *f (argument; parameters) → Result*
- *Name (arguments) { code }*

Functions: A Simple Example

```
def hello():
    print("Hello.")
```

- The function has a name.
- The function has a body. The body is delimited with space/tabs.
- We can invoke our function: `hello()`
- Note that print is itself a function.
- We will talk a lot more about Python functions later.

Functions Arguments

Functions: Arguments

```
def hello(greeting):
    print(greeting)
```

- A function can accept arguments (domain).
- Note the absence of argument type declarations.
  - Is that a good feature?
- We can now invoke our function with an argument.
  - `hello("Greetings!")`
  - `hello(5)`

Functions: Arguments Con't

```
def filter(my_list, x):
      for u in my_list:
            if u == x:
                  my_list.remove(x)
```

- Arguments are *positional*: order in invocation determines binding.
- Arguments can be simple scalars, objects or a mixture.
- Note that objects are passed by reference, that is, modifications made in a function will be visible to the caller.
  - This is by design. Function invocation is much faster.

Functions: Arguments Con't

```
def filter(my_list, x):
      for u in my_list:
            if u == x:
                  my_list.remove(x)

x = [1, 2, 3, 2, 4, 2, 5, 2, 6, 2]
filter(x, 2)       # x = [1, 3, 4, 5, 6]
filter(2, x)       # Run-time error
filter(x[:], 2)    # Passes a copy of x, x unaffected
```

Variable Number of Arguments

```
def post_fix(arguments):
     first_term = arguments[0]
     second_term = arguments[1]
     operation = arguments[2]
     …
values = [-5, 7, '+']
post_fix(values)
```

- We can support variable number of arguments by placing all of our arguments in a list.
- We then pass a single list as an argument.
- May require a convention so the list can be correctly interpreted.

Variable Number of Arguments

```
def post_fix(*operation):
    n = len(operation)
    for x in operation:
        print(x)

post_fix(-7, 5, '+')
post_fix(-7, 3, 4, '-','+')
```

- Python offers syntactic sugar for this pattern.
- Python does the work of building the list.

KWARGS: Keyword Arguments

```
def send(**msg):
        print(msg["from"], msg["to"], msg["data"])


Send(from="Doug", to="Sally", data=3.1415926)
# Doug Sally 3.1415926
```

- Conventions based on list position are cumbersome.
- A better approach is to name the arguments.
- Python dictionaries are just the thing.
- Python offers a calling convention to do the work for us.

Name Scope

```
def hypot(x, y):
        x = x * x                       # update local copies
        y = y * y
        return math.sqrt(x + y)

x = 5
y = 2
hypot (x, y)          # x and y will not be affected
```

- The scope of an identifier is where it is visible in a program.
- The scope of the function is private.
- Simple values, scalers, are passed by value (no side effects)

Name Scope

```
def hypot(z):
      z[0] = z[0] * z[0]              # Side effects
      z[1] = z[1] * z[1]
      return math.sqrt(z[0] + z[1])
w = [5, 2]
hypot(w)                    # w's elements are modified
hypot([5, 2])
```

- Arguments passed as a list, so they are mutable.
- Arguments used as an LVALUE will be globally modified.

Side Effects

```
def compute(x):
        s = x[:]                        # make a local copy
        do_work(s)                      # compute with copy

z = [5, 2, 8, 4, 9, 1, 5, 7]
compute(z)                              # z remains unchanged
compute(z[:])                           # pass a copy
```

- By default objects are passed by reference.
- In the event that side effects are not desired we can pass a copy.
  - Debugging
  - Experimenting

Scope Manipulation: `global`

```
alpha = 0.9
mu = 1.5
def f(x):
      global alpha                  # side effects
      mu = 2.0                      # hides external mu
      x = x * alpha * mu
      if x > 1.0:
            alpha = alpha * 0.5 # externally visible
            mu = mu - 1           # local only
```

- The Python reserved word `global` imports a name into the scope.

Scope Manipulation: `global`

```
alpha = 0.9
mu = 1.5
def f(x):
      global alpha                  # side effects
      # No need to declare mu, only used as an RVALUE
      x = x * alpha * mu
      if x > 1.0:
            alpha = alpha * 0.5 # externally visible
```

- We only need to declare an identifier if we use it as an LVALUE.

Scope Manipulation: `global`

```
def f(x, y, z):
      global debug                  # global scope
      debug = g(x, y, z)

print(debug) # debug is available outside of f
```

- We can also use global to export a local variable.
- Useful for debugging complicated routines (we can preserve and observe intermediate state externally).

Scope: `nonlocal`

```
def fib(n):                          # F_i = F_i-2 + F_i-1
    last = 0
    current = 1

    def sum():                       # Only in the scope of Fib
        nonlocal last                # Import last identifier, LVALUE
        nonlocal current             # Import current identifier, LVALUE
        f = last + current
        last = current               # LVALUE, side effects
        current = f

    for i in range(1, n + 1):
        sum()

    return current
```

- Python's `nonlocal` reserved word imports an identifier as an LVALUE.
- Useful for utility functions.

Function Return Values

Functions: Return a Value

```
def sum(x, y):
    return x + y
```

- Functions can return values (range).
- Notice that the declaration of the name, `sum`, did not include a type.
  - Is that a good feature?
- We can invoke our function thus:
  - `sum(1, 2) = 3`
  - `sum([1, 2], [3, 4, 5]) = [1, 2, 3, 4, 5]`
  - `sum("AI ", "is fun") = "AI is fun"`
  - `sum("Python", 5)`

A Function Can Return Anything

```
def expand(n):
    z = list()
    for i in range(0, n + 1):
        z.append(i)
    return z
```

- We can invoke our function thus:
  - `expand(5)` → `[0, 1, 2, 3, 4, 5]`

Returning a Tuple

```
def f(x):
    mean_x = statistics.mean(x)
    sd = statistics.stdev(x)
    return mean_x, sd
z = [1, 2, 3, 2, 4, 2, 5, 2, 6, 2]
y = f(z) → (2.9, 1.5951314818673865)
```

- We can return multiple values.
- Python will package the results in a tuple.
- We can access the results like a list:
  `y[1]` → `1.5951314818673865`

More on Tuples

```
def f(x):
      return 1, 2, (3, 4), 5


x = f()
x[1] → 2
x[2][0] → 3
```

- We can also nested tuples.
- The usual access rules apply.

Polymorphic Functions

```
def poly(x):
        if x > 0:
                return "I'm a string”
        elif x == 0:
                return                          # Returns None
        return 3.14159


poly(-1)
# 3.14159
poly(1)
# "I'm a string”
y = poly(0)
y == None → True
```

- A function can return any kind of object.
- Be careful: it is very easy to introduce a serious bug only found at run-time.
- The Python `type()` function can help you check if it matters.

Functions: Recursion

```
def factorial(n):      # N!
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

- A function can return the result of a function.
- The function invokes itself until it reaches the base case.
  - Can you spot the bug?
  - Can you spot the limitation?

Factorial: A Better Version

```
def factorial(n):
      if n < 0:
            return -1
      x = 1
      for i in range(1, n + 1):      # n inclusive
            x = x * i
      return x
```

- We check for an error.
- No chance of stack overflow.

next lecture:

- More on Functions
- Lambdas