ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Functions

- general form

```
def <name>(<parameter_list>):
   <statements>
   return <expression>
```

- creates a function object and assigns it to the given name
  - the optional `return` statement sends an object back to the caller
  - arguments are passed by *assignment*
  - no declarations of arguments, return types and local variables
- a simple example

```
def greet(friend):
   print('Hello {}!'.format(friend))
   return
```

- it is invoked using a *call expression*

```
greet(friend='world')
```

Scope and visibility

- the enclosing module acts as the global scope
- each call to the function creates a new local scope
- all assignments in the function body are local by default
  - you can override using the `global` statement
- all other names should either be global or built-in

```
root = 12

def isServer(pid):
   if pid == root: return True
   return False

def setServer(pid):
   global root
   root = pid
   return
```

Function arguments

- argument passing rules:
  - arguments are passed by creating a local reference to an existing object
  - re-assigning the local variable does not affect the caller
  - but modifying a mutable object through the reference impacts the caller
- argument matching modes:
  - by position
  - by keyword
  - using varargs:
    - $\star$: places non-keyword arguments in a tuple
    - $\star\star$: places keyword arguments in a dictionary
  - using default values supplied in the function declaration
- ordering rules:
  - declaration: normal, *arguments, **arguments
  - caller: non-keyword arguments first, then keyword arguments
- matching algorithm
  - assign non-keyword arguments by position
  - assign keyword arguments by matching names
  - assign left over non-keyword arguments to `*args`
  - assign extra keyword arguments to `**kwds`
  - unassigned arguments in declaration get their default values

Simple examples

- here are some examples of function declarations and the possible ways to invoke them – don't forget that python uses “pass by assignment”
- the simplest, but not the best, is positional invocation:

```python
# declare a function
def greet(friend): print('Hello {}!'.format(friend))

# invoke it
greet('world')
```

- a better way is to use the name of the dummy variable, as was done in our first example

```python
# declare a function
def greet(friend): print('Hello {}!'.format(friend))

# invoke it by explicitly binding the dummy variable to a value
greet(friend='world')
```

- it may look silly with one argument, but this technique eliminates more bugs than the strong type checking in languages like `C++`!
  - (long rant removed by the editor...)

Default arguments

- a function declaration can provide default values for arguments that were not provided by the caller; consider the following declaration:

```python
# declare a function
def say(what='Hello', whom='world'):
    print('{} {}!'.format(what, whom))
    return
```

- this function can be invoked as before

```python
# invoke with a full argument set -- note the twist
say(whom='cruel world', what='Goodbye')
```

- either one of the arguments can be absent; it will be bound to the default value

```python
# change the target
say(whom='class')
```

or

```python
# change the message
say(what='Greetings')
```

A potential pitfall

- there is some trickiness to default arguments, having to do with how this feature is currently implemented by the interpreter
  - the default values are treated as expressions that are evaluated during the function declaration, and stored along with the function itself
- consider the following function:

```python
# the mail routine
def mail(item, recipients=[]):
   # for each recipient
   for recipient in recipients:
      # mail the item
      print('mailing {} to {}'.format(item, recipient))
   # all done; return the recipient list back to the caller
   return recipients
```

- what happens when the return value is modified?

```python
# send a letter to the default set of recipients
friends = mail(item='a letter')
# add a couple of people to the list
friends += [ 'Alec', 'MacKenzie' ]
# send a postcard to the default set of recipients
mail(item='a postcard')
```

unlike what you might expect, the second invocation sends postcards to Alec and MacKenzie

Variable number of arguments

- occasionally – but rarely – there are legitimate reasons for a function to accept an unknown number of arguments
    - but please consider alternatives before resorting to this
    - we'll see a case where it is necessary when we discuss multiple inheritance
- consider:

```
# the mail routine
def mail(item, *recipients):
    # for each recipient
    for recipient in recipients:
        # mail the item
        print('mailing {} to {}'.format(item, recipient))
    # all done
    return
```

which can be invoked as

```
mail('letter', 'Alec', 'MacKenzie')
```

the variable `recipients` gets bound to a tuple of all the arguments that follow `'letter'`

Variable number of arguments

- it is now illegal to use the form `item='letter'` in the function call
- instead you gain the ability to do this:

```
friends = ['Alec', 'MacKenzie']
mail('letter', *friends)
```

- similar considerations apply to

```
# the mail routine
def configure(item, **options):
   print('configuring {!r}'.format(item))
   # for each option
   for option, value in options.items():
      # print the option
      print('  {} <- {}'.format(option, value))
   # all done
   return
```

which can be invoked using

```
# set some options
options = {
   'paper': 'A4',
   'orientation': 'landscape'
   }
# configure the printer
configure('printer', **options)
```

Functions as objects

- like everything else, functions are objects

```python
def hello():
    """say hello"""
    return "Hello"

def goodbye():
    """say goodbye"""
    return "Goodbye"

def greet(how=hello, whom='world'):
    """call {how} to compute what to say to {whom}"""
    print('{} {}!'.format(how(), whom))
    return

greet(how=goodbye, whom='class')
```

- the name of a function is a reference to the callable object that `def` left behind
  - you can use it anywhere a variable would go
  - to invoke the function, you must involve the reference in a call expression

```python
greeter = goodbye
message = greeter()
```