CIS192 Python Programming

Functional Programming

Robert Rand

University of Pennsylvania

February 03, 2016

Outline

1. Function Arguments
   - Positional and Named Arguments
   - Variable Number of Arguments
   - Variables Declared Outside Function

2. Functional Programming
   - Background
   - Higher Order Functions
   - Partial Application
   - Decorators

Positional Arguments

- `def func(arg1, arg2, arg3):`
  - arg1 arg2 and arg3 are positional arguments
  - When calling `func` exactly 3 arguments must be given
  - The order in the call determines which `arg` they are bound to
- `func(a, b, c)`
  - The expressions `a, b, c` are evaluated before the call
  - The value of `a` is bound to `arg1` in the body of func
  - Likewise `b` to `arg2` and `c` to `arg3`
  - Calling a function with the wrong number of args gives a `TypeError`

Named Arguments

- After the positional args, named args are allowed
- `def func(arg1, named1=val1, named2=val2):`
  - `named1` and `named2` are variables usable in the body of `func`
  - `val1` and `val2` are default values for those variables.
  - Omitting named arguments in a call uses the default value
- `func(a, named2=b, named1=c)`
  - named arguments can be given out of order
- `func(a, named2=b)`
  - The default value, `val1` will be bound to `named1`

Default Arguments

- Default arguments are evaluated when the function is defined
- In all calls, the object that the expression evaluated to will be used.
- If the default is mutable, updates in one call effect following calls
- `def func(a=[])` Will mutate the default on each call
- ```
  def func(a=None):
    if a is None:
      a = []
  ```
- Use `None` as the default to avoid mutation
  ```

Memoization

- Memoization is an optimization technique that stores results of function calls
- The previously computed answers can be looked up on later calls
- Use a dictionary default arg to store answers
- `def func(arg, cache={}):`
- Store answers in `cache[arg] = ans`
- Check for `arg in cache` before doing any work

*args

- A variable number of positional arguments can be specifed
- Use `*args` after all named args.
  - Could use any identifier but `args` is conventional
- `def func(arg1, *args)`
  - `args` is a tuple of 0 or more objects
- `func(a, b, c)`
  - `arg1 = a, args = (b, c)`

**kwargs

- A variable number of kwargs can be specifed
- Use `**kwargs` at the end
  - Could use any identifier but `kwargs` is conventional
- `def func(x, *args, **kwargs)`
  - `kwargs` is a dictionary of strings to values
  - The keys of `kwargs` are the names of the keyword args
- `func(7, 2, 4, exp=3, den=8)`
  - `x = a`
  - `args = (2,4)`
  - `kwargs = {'exp': 3, 'den':8}`

*/** in Function Definition or Assignment

- `def(*args)` args is a tuple that can take 0 or more values
- `def(**kwargs)` kwargs is a dictionary that can take 0 or more key-value pairs

*/** in Function Call

- `func(*expr)`
  - `expr` is an iterable
  - It gets unpacked as the positional arguments of `func`
  - Equivalently
    `seq = list(expr); func(seq[0], seq[1], ...)`
- `func(**expr)`
  - `expr` is a dictionary of form `{'string': val, ...}`
  - It gets unpacked as the keyword arguments of `func`
  - Equivalently `func('string'=val, ...)`

Closures

- AKA lexical closure or function closure
- A function that knows about variable defined outside the function
- ```
  a = 42
  def func():
    print(a)
  ```
- `func` is a closure because it knows about `a`
- Closures are read-only in Python
- ```
  a = 42
  def func():
    print(a)
    a += 1
  ```
- UnboundLocalError: local variable 'a' referenced before assignment

global

- `global` can circumvent read-only closures
- the `global` keyword declares certain variables in the current code block to reference the global scope
- ```
  a = 42
  def func():
    global a
    print(a)
    a += 1
  ```
- This does not raise an error
- Variables following `global` do not need to be bound already
  ```

Outline

Background

- Functional programming started with lambda(λ) calculus
  - Alternative to Turning machines for exploring computability
  - Expresses programs as functions operating on other functions
- Functional programming attempts to make it easier to reason about program behavior
  - Mathematical interpretation of functions allows mathematical proofs
- If data is immutable and there are no side-effects then functions always behave the same way
- Python data is mutable and allows side-effects
  - Has some functional concepts
  - Not an ideal functional programming environment

First Class Functions

- A higher order function is a function that:
  - Takes a function as one of its inputs
  - Outputs a function
- You can use functions anywhere you would use a value
- Functions are immutable so you can use them as dictionary keys
- Functions can be the return value of another function

λ (lambda) Functions

- Anonymous functions are function objects without a name
- `lambda arg: ret` is the same as
- ```
  def <lambda>(arg):
    return ret
  ```
- lambdas can have the same arguments as regular functions
  - `lambda arg, *args, named=val, **kwargs: ret`
- lambdas must be one-liners and do not support annotations
  ```

Higher Order Functions

- The most common are `map`, `filter`, and `reduce`(foldL)
- `map(f, seq)` returns an iterator containing each element of `seq` but with `f` applied
- `filter(f, seq)` returns an iterator of the elements of `seq` where `bool(f(seq[i]))is True`
- `filter(None, seq)` is the same as `filter(lambda x: x, seq)`
- `reduce` must be imported. `from functools import reduce`
- `reduce(f, seq, base)`
  - Builds up result by calling `f` on elements of `seq` starting with `base`
  - `f(...f(f(base,seq[0]),seq[1]),...)`
  - If `base` is not specified then the first argument is `seq[0]`
  - Calling `reduce` on an empty sequence is a `TypeError`

Functions as Keyword Args

- Many functions will accept another function as a kwarg
- `sorted(seq, key=f)`
  - `sorted` will call `f` on the elements to determine order
  - The elements in the resulting list will be the same objects in `seq`
  - Have the key return a tuple to sort multiple fields
- `min(seq, key=f)` and `max(seq, key=f)` behave similarly
- This is a good spot for `lambda`

Partial Application

- Partial application creates a new function by supplying an existing function with some of its arguments
- Say you have `add(x, y): x + y`
- You want `add_3(y): 3 + y`
- `add_3 = add(3)` raises a `TypeError`
- Use `from functools import partial`
- `add_3 = partial(add, 3)`

Decorators

- Decorators are transformations on functions
  - A function that takes in a function and returns a modified function

```
@dec
def func(arg1, arg2, ...):
  pass
```

- Is equivalent to

```
def func(arg1, arg2, ...):
  pass
func = dec(func)
```

Decorator Arguments

- A decorator can take arguments
- ```
  @decmaker(argA, argB, ...)
  def func(arg1, arg2, ...):
    pass
  ```
- Is equivalent to

  ```
  def func(arg1, arg2, ...):
    pass
  func = decmaker(argA, argB, ...)(func)
  ```
- `decmaker(argA, argB, ...)` returns a regular decorator
  ```

Multiple Decorators

- ```
  @dec1
  @dec2
  def func(arg1, arg2, ...):
    pass
  ```
- Is equivalent to

  ```
  def func(arg1, arg2, ...):
    pass
  func = dec1(dec2(func))
  ```
  ```