Information Systems
Laboratory Classes

Lia Morra Ph.D. lia.morra@polito.it
Eng. Mohammad Ghazi Vakili Mohammad.ghazivakili@polito.it
Eng. Sina Famouri Sina.famouri@polito.it

if...else as an expression

- There is a variant to the **if...else** statement: the **if...else** *expression*
- Use it for simple, one-line assignments, prints or something like that
  - ```
    x = 1 if some_condition() else 0
    print “x is 1” if x == 1 else “x is not 1”
    some_function(3 if x > 0 else 2)
    ```
- The value before the **if** is returned if the condition is true, otherwise the value after the **else** is returned
    ```

More on functions

- Default arguments
- Variable arguments
- Keyword arguments
- Argument unpacking
- Returning multiple objects

Default arguments

- Sometimes, we can provide meaningful default arguments to a function
  - This way, the caller doesn't need to provide those arguments
  - You can still provide your own arguments, if the default doesn't apply
- Default arguments are provided in the function's signature
  - ```
    # this will open a file, defaults to read-only mode
    def open_file(filename, mode='r'):
        return open(filename, mode)

    f = open_file('file.txt')
    f.read()
    f.close()
    f = open_file('file2.txt', 'w')
    f.write(data)
    f.close()
    ```
    ```

Variable arguments (1)

- Sometimes, you want a function to accept a variable number of arguments, because you don't know beforehand how many arguments will be provided
- You can do that in Python by using *args as a last parameter
  - ```
    def some_func(arg1, *args):
        print arg1
        print args

    some_func('hello', 1, 2, 'goodbye')
    ```
  - This will print:
    hello
    (1, 2, 'goodbye')
    ```

Variable arguments (2)

- The name “args” is just a variable name: you can call it whatever you want. The important part is the * (asterisk/star)
  - The convention is to call it “args”
- It MUST be the last argument in the function's signature
- All the extra parameters passed to the function will be put inside a tuple, which you can access in the function body
  - ```
    def some_func(arg1, arg2, *args):
        print arg1, arg2,
        for arg in args:
            print arg,

    some_func(1, 'abc', 'hello', 4, 5.3) # prints 1 abc hello 4 5.3
    ```
- It could also be an empty tuple if no extra parameters are passed
  - ```
    some_func(1, 'abc') # prints 1 abc
    ```

Keyword arguments (1)

- In Python, you don't have to pass the arguments to a function in the order they are expected: you can pass them by name
  - ```
    def some_func(a, b, c):
        print a, b, c

    some_func(1, 2, 3) → 1 2 3
    some_func(c=1, b=2, a=3) # prints 3 2 1
    ```
- This is especially useful if a function provides many default arguments, and you only want to override some
  - ```
    def some_func(name, age, address=None, phone=None, email=None):
        print name, age, address, phone, email

    some_func('Mario', 25) # prints Mario 25 None None None
    some_func('Mario', 25, email='mario@mail.it')
    # prints Mario 25 None None mario@mail.it
    ```

Keyword arguments (2)

- Your functions can also accept arbitrary keyword arguments using **kwargs as a last parameter
  - ```
    def some_func(arg1, arg2, **kwargs):
        print arg1, arg2
        print kwargs

    some_func(1, 2, a=3, b=4)
    ```
  - This will print:
    1 2
    { 'a': 3, 'b': 4 }
    ```

Keyword arguments (3)

- Keyword arguments are similar to variable arguments: the “kwargs” name is just a convention, the important part is the ** (double asterisk/star)
- It MUST be the last argument in the function's signature
- All the extra keyword arguments passed to the function are put inside a dictionary, where the argument name is the key, and its value the value
  - ```
    def some_func(**kwargs):
        for key, value in kwargs.items():
            print key, value, '|',

    some_func(a=1, b=2) # prints a 1 | b 2
    ```
- It could be an empty dictionary if no extra keyword arguments are passed
- You can use both variable arguments and keyword arguments in a function: in this case, keyword arguments must come AFTER variable arguments
  - ```
    def some_func(arg1, arg2, *args, **kwargs):
        pass
    ```

Argument unpacking (1)

- Variable and keyword arguments are very useful when calling a function and passing parameters directly
- What if my parameters are already inside a tuple, list or dictionary? And what if the function does not accept variable or keyword arguments?
  - Argument unpacking!

Argument unpacking (2)

- Using the * (star) and ** (double star) syntax, we can unpack a sequence and a dictionary respectively to populate the function's arguments
  - ```
    def my_func(a, b, c):
        print a, b, c

    #normal invokation
    my_func(1, 2, 3) # prints 1 2 3
    #with tuple unpacking
    parameters = (4,5,6)
    my_func(*parameters) # prints 4 5 6
    #with dictionary unpacking
    kwparams = { 'a': 3, 'b': 1, 'c':4 }
    my_func(**kwparams) # prints 3 1 4
    #this doesn't work, because the function wants 3 arguments
    my_func(parameters) # exception! my_func takes 3 arguments, 1
    provided
    my_func(kwparams) # exception! my_func takes 3 arguments, 1 provided
    ```
    ```

Argument unpacking (3)

- The same applies to functions that accept variable and/or keyword arguments
  - ```
    def my_fn(*args, **kwargs):
        print args, kwargs

    my_args = (1, 2)
    my_kwargs = { 'a': 3, 'b': 4 }
    my_fn(*my_args, **my_kwargs) # prints (1,2) { 'a':3, 'b':4 }
    #if you don't unpack, the result might not be what you expect
    my_fn(my_args, my_kwargs) # prints ((1,2), {'a':3, 'b':4}) {}
    #the 2 arguments are passed as variable arguments
    #they are put in a tuple, and the keyword arguments are empty
    ```
    ```

Argument unpacking (4)

- Unpacking is not only for functions: it works also with variables (implicitly)
  - ```
    a, b, c = (1, 2, 3) # a == 1, b == 2, c == 3
    a, b = 1, 2 # a == 1, b == 2
    x = (3,2,1)
    a, b, c = x # a == 3, b == 2, c == 1
    ```
- This is useful for swapping 2 variables in one line of code
  - ```
    a, b = 1, 2 # a == 1, b == 2
    a, b = b, a # a == 2, b == 1
    ```
- Note: these examples only use tuples, but unpacking works with any sequence: lists, strings, etc.
  - ```
    a, b, c = 'abc' # a == 'a', b == 'b', c == 'c'
    ```
    ```

Returning multiple objects

- Thanks to unpacking, we can return multiple objects from our functions
  - def some_func():
    return 1, 2, 3

    a, b, c = some_func() → a == 1, b == 2, c == 3
- And we can unpack the return value into another function call
  - def some_other_func(a, b, c):
    print a, b, c

    some_other_func(*some_func()) → 1 2 3

Functional features

- Functions as objects
- Callbacks
- Lambdas
- Closures
- Generator functions
- Decorator functions

Functions as objects

- In Python everything is an object. That includes functions as well
  - We can assign a function to a variable, then invoke it using the variable's name
- Examples:
  - ```
    def my_func():
        print 'hello'
    def func_with_params(a, b):
        print a, b
    def func_with_return(a):
        return a * 2

    fn = my_func
    fn2 = func_with_params
    fn3 = func_with_return

    fn() → hello
    fn2(1, 2) → 1 2
    print fn3(5) → 10
    ```
    ```

Callbacks (1)

- One way to exploit this feature is by using callback functions: functions that are called at some point by another function
  - def do_for_each_line(file, callback):
    line = file.readline()
    while line:
    callback(line)
    line = file.readline()

    def my_callback(line):
    print line

    def my_other_callback(line):
    print line[::-1]

    with open('file.txt', 'r') as file:
    do_for_each_line(file, my_callback) → this will print every line of the file

    with open('file2.txt', 'r') as file:
    do_for_each_line(file, my_other_callback) → this will print every line of the file backwards

Callbacks (2)

- With callbacks, we can define the general logic of an operation in one function, and let the callback function do the specific job
  - In the last example, do_for_each_line() simply implements a loop on each line of the file
  - What to do with each line is up to the callback function. This way, we can reuse do_for_each_line() in different contexts, making it do different things

Callbacks (3)

- There are many functions in Python or in external modules that exploit this concept
- filter(function, iterable) will return a list containing only the elements from iterable for which function(element) == True
  - In other words, it filters the input sequence using the function
  - def greater_than_zero(item):
    return item > 0

    my_list = [-1, 0, 2, -3, 5, 4, -4]
    my_filtered_list = filter(greater_than_zero, my_list)
    print my_filtered_list → [0, 2, 5, 4]

Lambdas (1)

- Very often, you will need a callback function that does something very simple, and you will use that function only once
- Instead of defining the function like we did in the previous examples, we can use **lambda functions**
- A lambda is simply an anonymous, inline, single-expression function
- The following 2 are equivalent
  - def greater_than_zero(x):
    return x > 0
  - lambda x: x > 0

Lambdas (2)

- The syntax for lambdas is simple: use the **lambda** keyword, followed by a list of arguments, then a : (colon) and finally an expression (the value of that expression will be returned automatically)
  - lambda x: x > 0
    lambda x, y: x + y == 3
    lambda z: z ** 2
- We can use lambdas instead of defining functions when we need a very simple callback function
  - my_list = [-2, 0, -4, 3, 6, -1]
    print filter(lambda x: x > 0, my_list) → [3, 6]

Scope

- Functions introduce what is known as a **scope**
- The scope of a function is the block of code where variables and functions are visible: out of the scope, they do not exist
  - def function():
    x = 1

    print x → exception! x is not defined

Closures (1)

- In Python, we can define a function inside another function
  - def outer():
        print 'outer func',
        def inner():
            print 'inner func'
        inner()

    outer() → outer func inner func

Closures (2)

- An inner function is declared in the scope of another function, therefore it is not visible outside that scope
  - def outer():
        def inner():
            pass

    outer() → ok!
    inner() → exception! inner is not defined

Closures (3)

- But functions are objects! So we can define an inner function and then return it
  - def outer():
        print 'I am the outer function'
        def inner():
            print 'I am the inner function'
        return inner

    x = outer() → I am the outer function
    x() → I am the inner function

Closures (4)

- What happens if we return an inner function that references variables that are visible from its scope, but not from the outside?
  - def outer():
    x = 3
    def inner(y):
    return x * y
    return inner

  x = outer()
  x(5) → what do you think will happen?

Closures (5)

- When something like that happens, Python “remembers” the scope in which the inner function was defined
- Even if that scope is not accessible from the outside, the inner function can still access it
- This “remembered scope” is called a **closure** and is a very powerful feature

Closures (6)

- Example: generating a power function
  - def power_of(exp):
        def inner(base):
            return base ** exp
        return inner

```
pow_2 = power_of(2)
pow_4 = power_of(4)
print pow_2(5), pow_2(8) # prints 25 (5**2) and 64 (8**2)
print pow_4(2), pow_4(5) # prints 16 (2**4) and 625 (4**4)
```

Generator functions (1)

- Closures can be used to remember a context between function calls
  - Basically they behave like object methods: they can remember their state
- A generator function is a function that automatically returns a generator object
- Generator objects have a .next() method that every time is invoked will return the next value
- Generators are used usually to generate sequences

Generator functions (2)

- Example: generator that counts up to N
  - ```
    def count_to(n):
        i = 0
        while i < n:
            yield i
            i += 1

    for n in count_to(10):
        print n
    ```
    ```

Generator functions (3)

- Generators are defined the same as functions. The only difference is that they don't use the `return` keyword but `yield`.
  - A function with the `return` keyword is a normal function
  - A function with the `yield` keyword is a generator
- The `yield` keyword behaves like `return` because it provides a value back to the caller
- The big difference is that when the generator is invoked a second time, execution continues from the line after `yield`

Generator functions (4)

- def demo_generator():
    print 'First call'
    yield 1
    print 'Second call'
    yield 2
    print 'Third call'
    yield 3

  gen = demo_generator()
  gen.next() → prints 'First call' and returns 1
  gen.next() → prints 'Second call' and returns 2
  gen.next() → prints 'Third call' and returns 3
  gen.next() → raises StopIteration exception

Generator functions (5)

- Generators are useful because they occupy little memory and can potentially generate an infinite amount of results
- In Python 2.7.x, the `range()` function returns a list of integers
  - You can use the `xrange()` generator (it has the same signature as `range()`) to obtain a generator that will yield the same sequence of numbers
  - Useful for iterating:
    - `for i in range(1000000000)`
      - This line will create a list with 1000000000 numbers → that's more than 1GB of memory!
      - Possibly you could get a MemoryError (not enough memory for the whole list)
    - `for i in xrange(1000000000)`
      - This line will iterate 1000000000 times, but will only occupy a few bytes (the size of an integer holding the current value)

Decorators (1)

- You've seen how it's possible to pass a function as an argument of another function
- You've also seen you can return a function
- These two concepts are used to implement decorators
- A decorator is a piece of code that “decorates” (i.e. changes the behavior) of another function

Decorators (2)

- Main idea: a decorator is a function that takes another function as argument and returns a new, “decorated” function
- The new function should look and behave exactly like the original, plus the “decoration”
- This way, it's possible to substitute the old function with the new one

Decorators (3)

- Example:
  - ```
    def test_deco(fn_to_decorate):
        def inner():
            print 'Before'
            fn_to_decorate()
            print 'After'
        return inner

    def test_fn():
        print 'Hello world'

    decorated_fn = test_deco(test_fn)
    decorated_fn()
    ```
- The example code will print 'Before', then execute the test_fn function (and print 'Hello world'), and then print 'After'
    ```

Decorators (4)

- The logic here is simple:
  - We have an arbitrary function (fn_to_decorate)
  - We define a new function (inner) with the same signature as fn_to_decorate
    - This way, the original function and the decorated one look exactly the same and can be used the same way
    - The new function adds behavior, and at some point calls the original function
  - Finally, we return the new function to the caller

Decorators (5)

- In order to support any function, decorators usually forward the *args and **kwargs to the original function. This way they can be used with any function
  - ```
    def test_deco(fn):
        def inner(*args, **kwargs):
            print 'before'
            fn(*args, **kwargs)
            print 'after'
        return inner

    def test_fn2(x, y):
        print x + y

    decorated_fn = test_deco(test_fn2)
    decorated_fn(1, 2) → prints 'before', 3 and 'after'
    ```
    ```

Decorators (6)

- Decorators are such a common pattern in Python that a special syntax was introduced
  - By writing `@decorator_name` just before a function declaration, that function is automatically decorated by `decorator_name`
- The @ syntax is just for convenience
  - ```
    @test_deco
    def test():
        pass
    ```
  - is exactly the same as
  - ```
    def test():
        pass
    test = test_deco(test)
    ```
- The @ syntax substitutes the decorated function with the return value of the decorator

Decorators (7)

- Example: add 1 to the result
  - ```
    def off_by_one(fn):
        def inner(*args, **kwargs):
            return fn(*args, **kwargs) + 1
        return inner

    @off_by_one
    def add(x, y):
        return x + y

    add(3, 5) → returns 9 (3 + 5) + 1 (due to the decorator)
    ```
    ```

Decorators with arguments (1)

- You can also pass arguments to a decorator in order to customize it
- However, the @ notation expects a function that accepts only one argument (the function to decorate) and returns a new function
- Therefore, we have to return a “standard” decorator depending on the argument

Decorators with arguments (2)

- Example: add N to the result
  - ```
    def off_by_n(n):
        def std_deco(fn):
            def inner(*args, **kwargs)
                return fn(*args, **kwargs) + n
            return inner
        return std_deco

    @off_by_n(5):
    def add(x, y):
        return x + y

    add(1, 2) → returns 3 + 5 (due to the decorator)
    ```
    ```