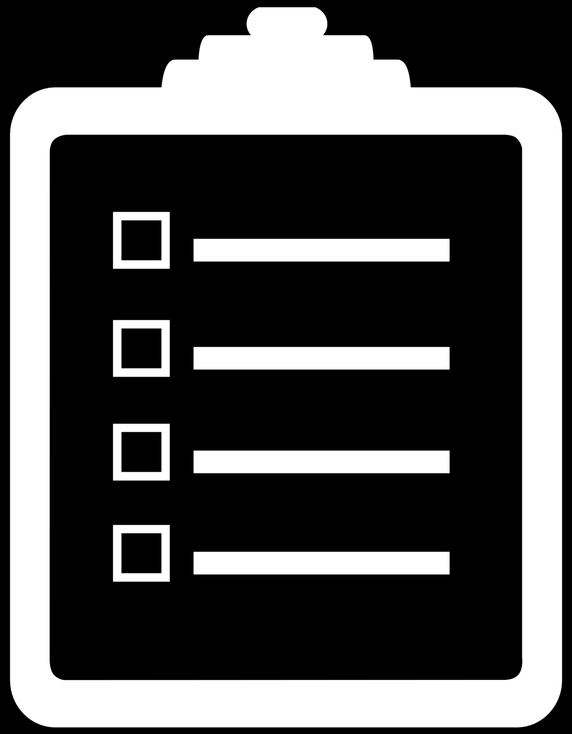
Functions

April 17, 2017

Agenda



- Review of Last Week
- Inside Python Functions
- Keyword Arguments
- Variadic Arguments
- First-Class Functions

Recap

Data Structures

Lists `[items]`

Dictionaries `{key: value}`

Tuples `(frozen, sequence)`

Sets `{unique, hashable, values}`

Comprehensions `[f(xs) for xs in iter]`

Warming Up: 10 Minutes

Write an program that, given letters, finds all anagrams

```
$ python anagrammer.py
Letters? hnopty
Anagram(s): ["python"]
Letters? sulpcpusl
Anagram(s): []
```

Key insight: The sorted letters of any two anagrams are the same: {sorted letters: list of anagrams}

Familiar Functions

Recall

The **`def`** keyword is used to define a new function

```
def fn_name(param1, param2):
    value = do_something()
    return value
```

Basic Functions: Nuances

Return

All functions return *some* value

- Even if that value is **`None`**

No **`return`** statement or just **`return`** implicitly returns **`None`**

The interpreter suppresses printing **`None`**

Returning multiple values

- You can use a tuple! In some cases, use a **`namedtuple`**

```
return value1, value2, value3
```

Be careful! Callers may not expect a tuple as a return value

Function Execution and Scopes

Function execution introduces a new local symbol table (scope)

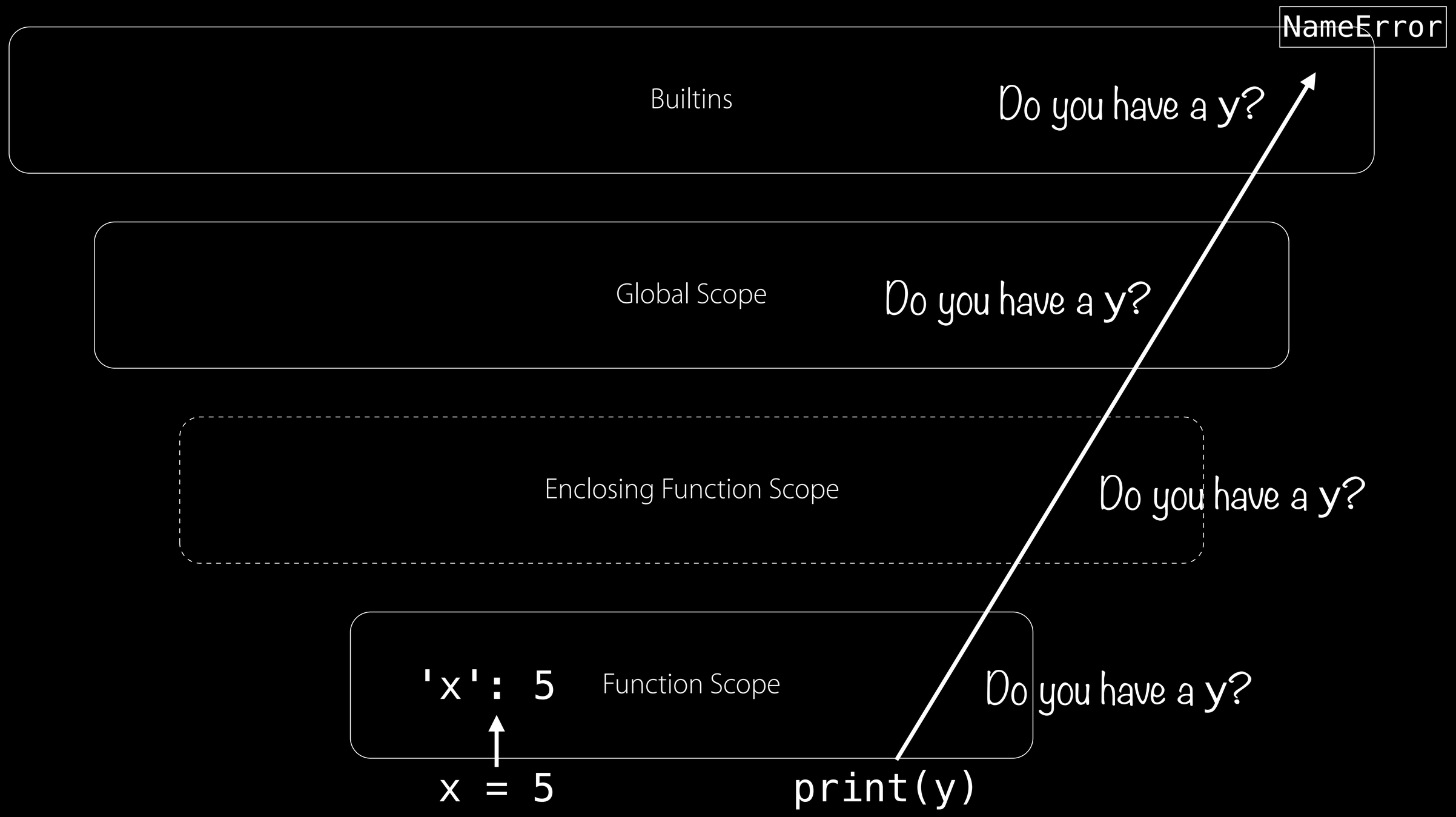
- Think of baggage tags and suitcases: a new baggage area

Variable assignments (L-values) `x = 5`

- Add entry to local symbol table (or overwrite an existing entry)

Variable references (R-values) `print(y)`

- First, look in local symbol table
- Next, check symbol tables of enclosing functions (unusual)
- Then, search global (top-level) symbol table
- Finally, check builtin symbols (`print`, `input`, etc)


NameError
Builtins
Do you have a y?
Global Scope
Do you have a y?
Enclosing Function Scope
Do you have a y?
'x': 5
Function Scope
Do you have a y?
x = 5
print(y)

Local Function Scope

```
x = 2
def foo(y):
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
# prints {'y': 3, 'z': 5}
# print 2
# prints 2, 3, 5
```

Local Function Scope

```
x = 2
def foo(y):
    x = 41
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
# prints {'x': 41, 'y': 3, 'z': 5}
# print 2
# prints 41, 3, 5
```

We've added an `'x': 41` entry to the local symbol table

If / For Scope

Notably, only* function definitions define new scopes

`if` statements, `for` loops, `while` loops, `with` statements, etc

Do *not* introduce a new scope

```
if success:
    desc = 'Winner!'
else:
    desc = 'Loser :('
print(desc)
```

*Also classes... kinda (Wk 5)

Pass-By-Value or Pass-By-Reference?

Variables *are* copied into function's local symbol table

- But variables are just references to objects!

Best to think of it as *pass-by-object-reference*

- If a mutable object is passed, caller will see changes

Baggage tags in one area can point to suitcases in another

Default Parameters

Default / Named Parameters

Specify a default value for one or more parameters

- Called with fewer arguments than it is defined to allow

Usually used to provide "settings" for the function.

Why?

- Presents a simplified interface for a function
- Provides reasonable defaults for parameters
- Declares intent to caller that parameters are "extra"

Required parameter `prompt`

```
def ask_yn(prompt,
           retries=4,
           complaint='...'):
```

Optional parameter `retries` defaults to 4

Optional parameter `complaint` defaults to 'Enter Y/N'

Keyword Arguments

```
def ask_yn(prompt, retries=4, complaint='Enter Y/N!'):
    for i in range(retries)
        ok = input(prompt)
        if ok == 'Y':
            return True
        if ok == 'N':
            return False
        print(complaint)
    return False
```

Examples

```
print(…, sep=' ', end='\n', file=sys.stdout, flush=False)
range(start, stop, step=1)
enumerate(iter, start=0)
int(x, base=10)
pow(x, y, z=None)
seq.sort(*, key=None, reverse=None)

subprocess.Popen(args, bufsize=-1, executable=None,
stdin=None, stdout=None, stderr=None, preexec_fn=None,
close_fds=True, shell=False, cwd=None, env=None,
universal_newlines=False, startupinfo=None,
creationflags=0, restore_signals=True,
start_new_session=False, pass_fds=())
```

Wow...

ask_yn(prompt, retries=4, complaint='...')

```
# Call with only the mandatory argument
ask_yn('Really quit?')

# Call with one keyword argument
ask_yn('OK to overwrite the file?', retries=2)

# Call with one keyword argument - in any order!
ask_yn('Update status?', complaint='Just Y/N')

# Call with all of the keyword arguments
ask_yn('Send text?', retries=2, complaint='Y/N please!')
```

Dead Parrot

```
def parrot(voltage, state='a stiff', action='voom',
           type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Valid Calls

```
def parrot(voltage, state='…', action='…', type='…')

# 1 positional argument
parrot(1000)
# 1 keyword argument
parrot(voltage=1000)
# 2 keyword arguments
parrot(voltage=1000000, action='VOOOOOM')
# 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)
# 3 positional arguments
parrot('a million', 'bereft of life', 'jump')
# 1 positional, 1 keyword
parrot('a thousand', state='pushing up the daisies')
```

Invalid Calls

```
def parrot(voltage, state='…', action='…', type='…')

# required argument missing
parrot()
# non-keyword argument after a keyword argument
parrot(voltage=5.0, 'dead')
# duplicate value for the same argument
parrot(110, voltage=220)
# unknown keyword argument
parrot(actor='John Cleese')
```

Rules about Function Calls

Keyword arguments must follow positional arguments

All keyword arguments must identify some parameter

- Even positional ones

No parameter may receive a value more than once

```
def fn(a): pass
fn(0, a=0)
# Not allowed! Multiple values for a
```

Variadic Positional Arguments

Variadic Positional Arguments

A parameter of form `*args` captures excess positional args

- These excess arguments are bundled into an `args` tuple

Why?

- Call functions with any number of positional arguments
- Capture all arguments to forward to another handler
  - Used in subclasses, proxies, and decorators

```
print(*objects, sep=' ', end='\n', file=…, flush=False)
```

Variadic Positional Arguments

```
# Suppose we want a product function that works as so:
product(3, 5)  # => 15
product(3, 4, 2)  # => 24
product(3, 5, scale=10)  # => 150


# product accepts any number of arguments
def product(*nums, scale=1):
    p = scale
    for n in nums:
        p *= n
    return p
```

Named parameters after `*args` are 'keyword-only' arguments (why?)

Unpacking Variadic Positional Arguments

```
# Suppose we want to find 2 * 3 * 5 * 7 * … up to 100
def is_prime(n):  pass  # Some implementation

# Extract all the primes
primes = [number for number in range(2, 100)
          if is_prime(number)]

# primes == [2, 3, 5, …]
print(product(*primes))  # equiv. to product(2, 3, 5, …)
```

The syntax `*seq` unpacks a sequence into its constituent components

Variadic Keyword Arguments

Variadic Keyword Arguments

A parameter of the form `**kwargs` captures all excess keyword arguments

- These excess arguments are bundled into a `kwargs` dict

Why?

- Allow arbitrary named parameters, usually for configuration
- Similar: capture all arguments to forward to another handler
  - Used in subclasses, proxies, and decorators

Variadic Keyword Arguments

```
authorize(
    "If music be the food of love, play on.",
    playwright="Shakespeare",
    act=1,
    scene=1,
    speaker="Duke Orsino"
)

# > If music be the food of love, play on.
# ----------------------------------------
# act: 1
# scene: 1
# speaker: Duke Orsino
# playwright: Shakespeare
```

Variadic Keyword Arguments

```
def authorize(quote, **speaker_info):
    print(">", quote)
    print("-" * (len(quote) + 2))
    for k, v in speaker_info.items():
        print(k, v, sep=': ')
```

```
speaker_info = {
  'act': 1,
  'scene': 1,
  'speaker': "Duke Orsino",
  'playwright': "Shakespeare"
}
```

Unpacking Variadic Keyword Arguments

```
info = {
    'sonnet': 18,
    'line': 1,
    'author': "Shakespeare"
}

authorize("Shall I compare thee to a summer's day", **info)

# > Shall I compare thee to a summer's day
# ----------------------------------------
# line: 1
# sonnet: 18
# author: Shakespeare
```

Example: Formatting Strings

All positional arguments go into `args`

```
fstr.format(*args, **kwargs)
```

All keyword arguments go into `kwargs`

fstr.format(*args, **kwargs)

```
# {n} refers to the nth positional argument in `args`
"First, thou shalt count to {0}".format(3)
```

```
args = (3, )
```

```
"{0} shalt thou not count, neither count thou {1},
excepting that thou then proceed to {2}".format(4, 2, 3)
```

```
args = (4, 2, 3)
```

```
# {key} refers to the optional argument bound by key
"lobbest thou thy {weapon} towards thy foe".format(
    weapon="Holy Hand Grenade of Antioch"
)
```

```
kwargs = {"weapon": "Holy Hand Grenade of Antioch"}
```

Complicated Example

```
"{0}{b}{1}{a}{0}{2}".format(
    5, 8, 9, a='z', b='x'
)
# => 5x8z59
```

```
args = (5, 8, 9)
kwargs = {'a':'z', 'b':'x'}
```

Cute Trick: Unpacking Variadic Keyword Arguments

```
x = 3
foo = 'fighter'
y = 4
bar = 'bell'
z = 5
```

local symbol table

```
{
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'bar': 'bell',
    'z': 5, …
}
```

```
print("{z}^2 = {x}^2 + {y}^2".format(x=x, y=y, z=z))
print("{z}^2 = {x}^2 + {y}^2".format(**locals()))
# Equivalent to .format(x=3, foo='fighter', y=4, ...)
```

Usually slow… and bad style, but can be useful for debugging!

Putting it All Together

A Valid Python Function Definition

Mandatory positional arguments

Variadic positional argument list – scoops up excess positional args into a tuple

```
def foo(a, b, c=1, *d, e=1, **f)
```

Optional keyword argument

Optional keyword-only argument

Variadic keyword argument list – scoops up excess keyword args into a dictionary

Time Out for Announcements

Logistics

Office Hours

Sam (after class/by appointment), Course Staff (GCalendar)

Assignment 1

Cryptography (Caesar, Vigenere, Merkle-Hellman)

Enrollment

Still movement on the waitlist… stay tuned!

Lab Solutions

Back to Python!

Aside: Code Style

Function Comments

- The first string literal *inside* a function body is a docstring
  - First line: one-line summary of the function
  - Subsequent lines: extended description of function
- Describe parameters (value / expected type) and return
  - Many standards have emerged (javadoc, reST, Google)
  - Just be consistent!
- The usual rules apply too! List pre-/post-conditions, if any.

Example: Function Docstrings

```
def my_function():
    """Summary line: do nothing, but document it.

    Description: No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)

# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.
```

More: PEP 257

General Good Practices

Spacing Use 4 spaces to indent. Don't use tabs.

Use blank lines to separate functions and logical sections inside functions.

Use spaces around operators and after commas, but not directly inside delimiters

```
a = f(1, 2) + g(3, 4)
```

Commenting Comment all nontrivial functions.

Add header comments at the top of files before any imports.

If possible, put comments on a line of their own.

Naming Use snake_case for variables and functions (CamelCase for classes)

Decomposition and Logic Same as in 106s

More: PEP 8

Remember the Zen of Python

First-Class Functions

First-Class Functions

```
def echo(arg): return arg

type(echo)  # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)  # <function echo at 0x1003c2bf8>

foo = echo
hex(id(foo))  # 0x1003c2bf8
print(foo)  # <function echo at 0x1003c2bf8>

isinstance(echo, object)  # => True
```

Functions are Objects

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameters to other functions?

Can a function return another function?

How can I modify a function object?

WE MUST GO DEEPER (lab)

Summary

Reference

All functions return *some* value (possibly None)

Functions define scopes via symbol tables

Parameters are passed by object reference

Functions can have optional keyword arguments

Functions can take a variable number of args and kwargs

Use docstrings and good style

Functions are objects too (?!)