Warmup

Write a program that, given a list of letters, finds all anagrams.

```
$ python anagrammer.py
Letters? unicorn
Anagram(s): UNICORN
Letters? python
Anagram(s): PYTHON, PHYTON
Letters? aoiuvbosidbvoahbs
Anagram(s): None
```

A key insight: the sorted letters of any two anagrams is the same. Consider making a dict `{sorted letters: set of words}`

Functions

January 22, 2020

Week 3 of CS 41

Today in CS41:

- Review of Data Structures
- Advanced Looping
- Comprehensions
- Inside Python Functions
- Parameters
- Variadic Arguments
- Parameter Ordering
- Aside: Code Style
- First-Class Functions

Review of Data Structures

Data Structures

List	`[whatever, objects, you, want]`
Tuple	`(frozen, sequence, of, objects)`
Dictionary	`{key: value}`
Set	`{unique, hashable, values}`

Advanced Looping

zip

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}? It is {1}.'.format(q, a))


# What is your name? It is lancelot.
# What is your quest? It is the holy grail.
# What is your favorite color? It is blue.
```

Generates pairs of entries from its arguments

sorted

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange',
          'banana']
for f in sorted(set(basket)):
    print(f)

# apple
# banana
# orange
# pear
```

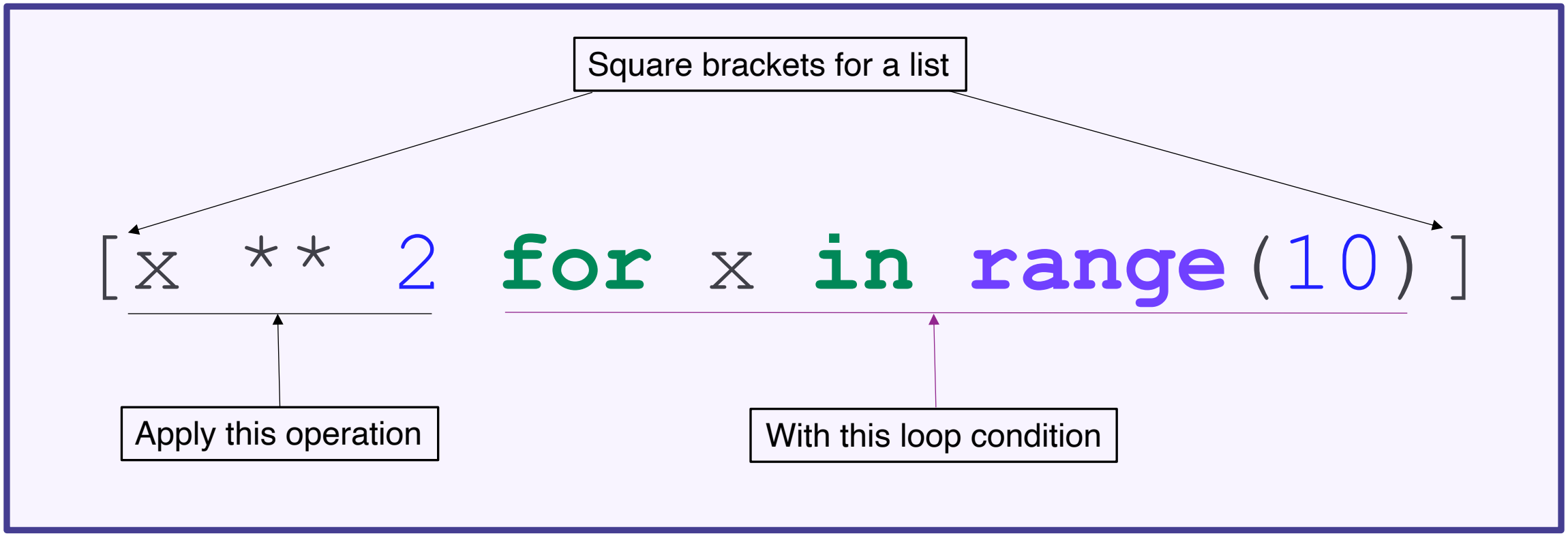
Returns a new sorted list while leaving the source unaltered

Comprehensions

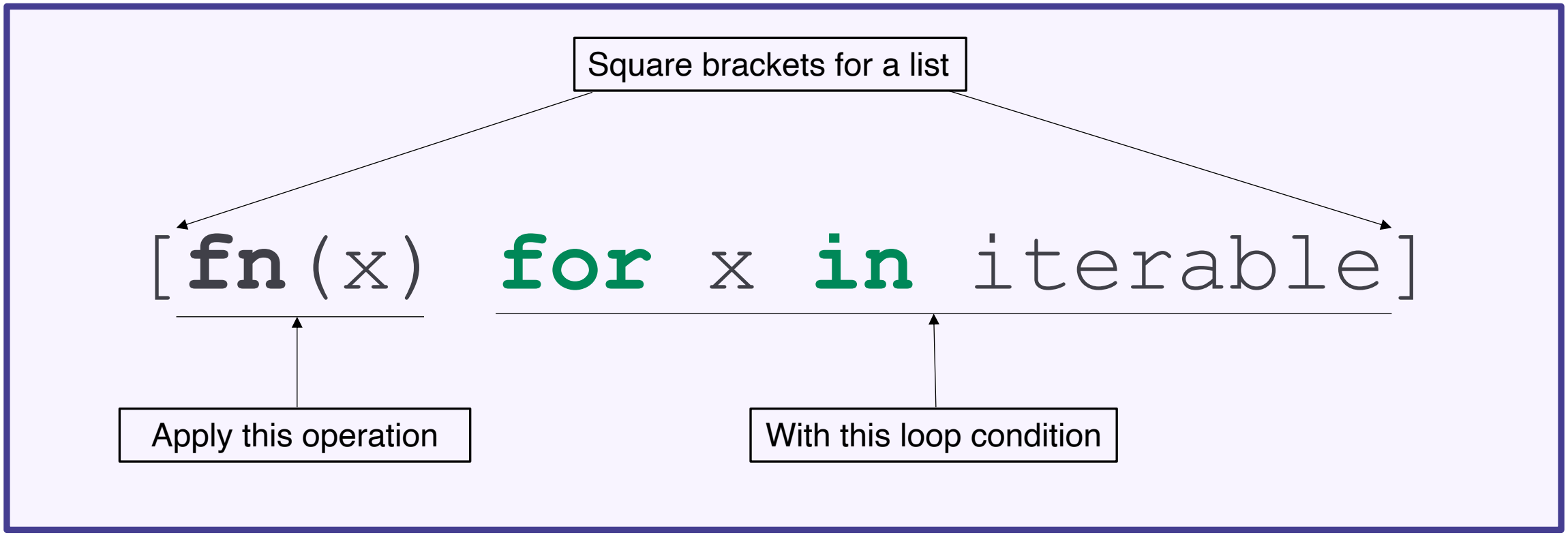
Square Numbers

```
squares = []
for x in range(10):
    squares.append(x**2)

squares # => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Square Numbers (Comprehension)



List Comprehensions



List Comprehensions

```
[fn(x) for x in iterable if cond(x)]
```

Only keep elements that satisfy a boolean condition

Examples

```
[word.lower() for word in sentence]
[ch for ch in word.lower() if ch not in 'aeiou']

[(x, x**2, x**3) for x in seq]
[(i,j) for i in range(5) for j in range(i)]
```

Be careful! Simple is better than complex.

Your turn!

```
[0, 1, 2, 3] -> [1, 3, 5, 7]
[3, 8, 9, 5] -> [True, False, True, False]

['apple', 'orange', 'pear'] -> ['A', 'O', 'P']
['apple', 'orange', 'pear'] ->
    [('apple', 5), ('orange', 6), ('pear', 4)]
```

Other Comprehensions

```
# Dictionary comprehensions
{key_fun(x):val_fun(x) for x in iterable}
fav_animals = {
    'parth': 'unicorn',
    'michael': 'elephant',
    'zheng': 'tree',
    'sam': 'ox',
    'nick': 'Daisy'
}
fav_humans = {val:key for key, val in fav_animals.items()}

# Set comprehensions
{fun(x) for x in iterable}
```

A High-Level View

Why would we use comprehensions?

Usual focus: Modify individual elements of an iterable.

Comprehensions ~ Abstract Transformations

Don't say how you want the object constructed.

Just say what you want *in* the object.

If you liked this, stay tuned… Functional Programming!

Inside Python Functions

So far…

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

```
def fn_name(param1, param2):
    ...
    return some_value
```

Some nuances…

All functions return *a* value…

If not specified or just "`return`" implicitly, functions return `None`.

Returning multiple values…

You can use a tuple!

```
return val1, val2, val3
```

Be careful! Sometimes callers might not expect a tuple as a return value. Using a `namedtuple` may be useful.

Function Execution and Scope

Function execution introduces a new local *symbol table* (literally: a `dict` that associates names to values).

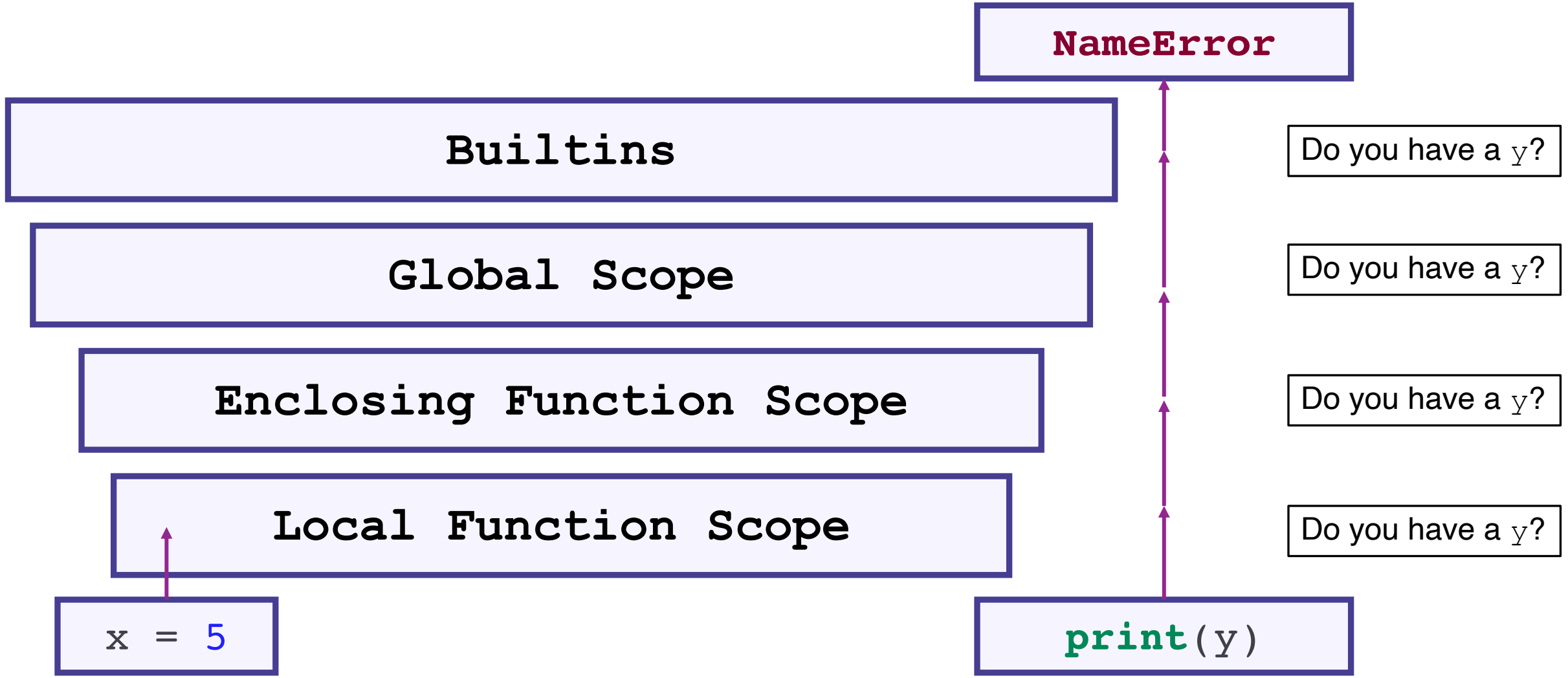
Variable assignments add to the local symbol table (or overwrite existing values).

```
x = 5
```

Variable references check a tower of scopes.

```
print(y)
```

Variable Resolution



Function Scopes: An Example

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


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

Function Scopes: An Example

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

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

Scope vs. Namespaces

A namespace is a dictionary, mapping names (as strings) to values.

A scope defines which namespaces will be looked in and in what order.

So, scopes can be overlapping (a global variable can be reached by any local reference), but namespaces are not overlapping.

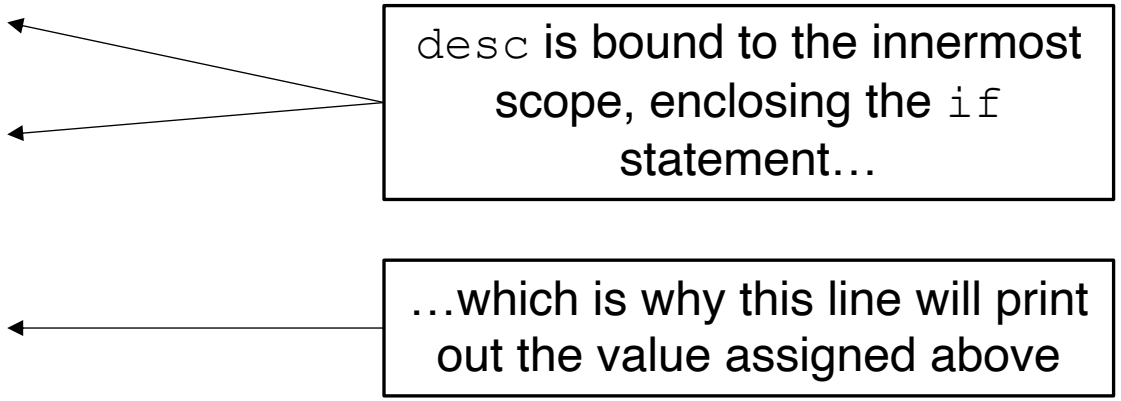
Only* function definitions define new scopes/namespaces.

*And classes; classes are more like *wrappers* around namespaces. Stay tuned for more!

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)
```



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.

Pass-by-Value or Pass-by-Reference?

IMMUTABLE

```
x = 5
x += 1
x # => 6
```

A *new* object is created and rebound to the namespace as `x`

```
def foo(x):
    x += 1

x = 5
foo(x)
x # => 5
```

A *new* object is created and rebound, but `x` only scopes over `foo`, so it only updates locally.

MUTABLE

```
x = [5]
x.append(41)
x # => [5, 41]
```

No new object is created. The value of `x` is modified…

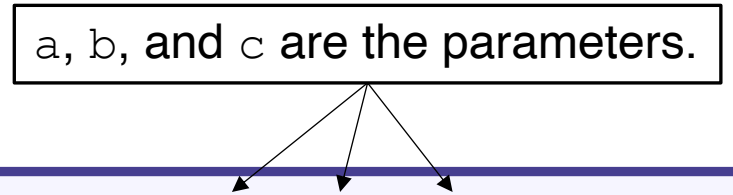
```
def foo(x)
    x.append(41)

x = [5]
foo(x)
x # => [5, 41]
```

Because `x` is not being rebound, changes will propagate.

Parameters

So far…

So far, we've seen *required positional parameters*.




```
def compute(a, b, c):
    return (a + b) * c
```

(they're required)

Default Parameters

Specify a default value for one or more parameters.

- Can be called with fewer arguments than it has!

Usually to provide “settings” for the function and specify certain default values.

Why?

- Presents a simplified interface.
- Provides reasonable defaults for parameters.
- Declares that certain parameters are “extra.”

Default Parameters

```
def ask_yn(prompt,
           retries=4,
           complaint="Enter Y or N!"):
```

`prompt` is required

`retries` is optional, defaults to `4`

`complaint` is also optional, defaults to `"Enter Y or N!"`

```
# A valid call:
ask_yn('Ok to overwrite the file?', 2)
```

`prompt`

Overwrite default value of `retries`

Default Arguments

```
def ask_yn(prompt, retries=4, complaint="Enter Y or N!"):
    for i in range(retries):
        ans = input(prompt)

        if ans in 'yY':
            return True
        if ans in 'nN':
            return False

        print(complaint)
```

Within the function: treat the parameters just like always!

Keyword Arguments

```
def ask_yn(prompt, retries=4, complaint="Enter Y or N!"):
```

All of the above parameters can be called based on position or as *keyword arguments*.

- Specify the name of the variable in the call.

Why?

- Make parameter intent clearer – name conveys more than position.
- Reduce risk of incorrect calls.

Keyword Arguments

```
def ask_yn(prompt, retries=4, complaint="Enter Y or N!"):
```

```
# Valid calls:
ask_yn('Ok to overwrite the file?', 2)
ask_yn('Ok to overwrite the file?', retries=2)
ask_yn(prompt="Are you sure about that?", retries=2)
ask_yn(retries=2, prompt="Are you sure about that?")
```

Order does not matter for these calls!

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=None, startupinfo=None,
    creationflags=0, restore_signals=True,
    start_new_session=False, pass_fds=(), encoding=None,
    errors=None, text=None)
```

Wow…

Variadic Arguments

Variadic Positional Arguments

A parameter of the form `*args` captures excess positional arguments.

- These arguments are bundled into the `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=sys.stdout, flush=False)
```

Variadic Positional Arguments

```
# Suppose we want a function that works as so:
scaled_sum(1, 2, 3)          # => 6
scaled_sum(1, 5)             # => 6
scaled_sum(1, 5, scale=10) # => 60
# i.e., scaled_sum(x1, ..., xn, scale=a) = a(x1 + ... + xn)

# scaled_sum accepts any number of arguments
def scaled_sum(*args, scale=1):
    return scale * sum(args)
```

Unpacking Variadic Positional Arguments

```
# Suppose we want to find the sum of the primes up to 100
def is_prime(p): ... # some implementation


primes = [p in range(1, 100) if is_prime(p)]


print(scaled_sum(*primes))
# equivalent to print(scaled_sum(2, 3, 5, ...))
```

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.

```
"{good} are better than {bad}".format(good='Reindeers', bad='people')
```

Variadic Keyword Arguments

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

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

Variadic Keyword Arguments

```
def stylize_quote(quote, **speaker_info):
    print("> {}".format(quote))
    print('-' * (len(quote) + 2))

    for k, v in speaker_info.items():
        print("{}: {}".format(k, v))
```

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

Unpacking Variadic Keyword Arguments

```
info = {
    'speaker': 'Iron Man',
    'year': 2012,
    'movie': 'The Avengers'
}

stylize_quote("Doth mother know you weareth her drapes?", **info)

# > Doth mother know you weareth her drapes?
# ----------------------------------------
# speaker: Iron Man
# year: 2012
# movie: The Avengers
```

Example: Formatting Strings

String Formatting

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

str.format(*args, **kwargs)

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

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

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

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

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

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

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

str.format(*args, **kwargs)

```
# A cute trick…
x = 3
foo = 'fighter'
y = 4
learn = 2, 'fly'
z = 5
```

```
locals() == {
    'x': 3,
    'foo': 'fighter',
    'y': 4,
    'learn': (2, 'fly'),
    '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, ...)
```

This is kind of how fstrings work!

Halftime

Logistics

Assignments	A0 and A1
Enrollment	Waitlist spots!
Labs	Labelled by Week (Lab 2 ~ Week 2)
	No lab this week. Lab will happen next week!
Office Hours	Cancelled on Friday

Assignment 1

- Cryptography!
  - Your assignment has *secrets within it*!
  - Hint: Think about this class…
- Implement different ciphers: Caesar, Vigenère, MHKC
- Practice data structures
- Due **Febuary 4, 2020 at 11:59pm**

Parameter Ordering

Parameter Rules

1. Keyword arguments must follow positional arguments.
2. All arguments must identify some parameter.
   Even positional ones.
3. No parameter may receive a value more than once.

Parrots Go VOOM

```
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, "!")

parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='VOOOOOM')
parrot(action='VOOOOOM', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

Parrots Go VOOM: Incorrect Calls

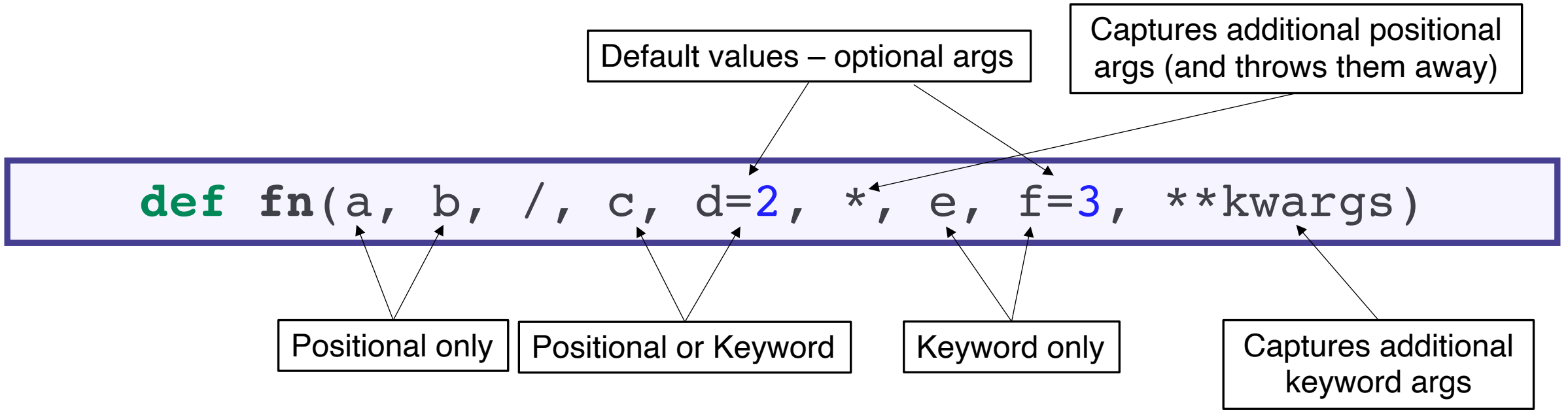
```
def parrot(voltage, state='a stiff', action='voom',
           type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")

# 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')
```

Parameter Rules

Symbols

`/`	New in Python 3.8: Arguments before this are positional-only. PEP 570
`*`	Arguments before this are positional or keyword arguments. Arguments after this are keyword only. (Making sure to follow rule 1)
`*a`	Captures trailing positional arguments.
`**kw`	Captures extra keyword arguments.

Parameter Rules



Note: it would not be valid if called with `c` keyword and `d` positional!

You'll evaluate valid/invalid calls in lab. Use this slide as a reference!

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 (value / type).
  - Many standards have emerged (javado, rest, Google)
  - Just be consistent!
- The usual rules apply too! List pre-/post-conditions, if any.

Example: 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) # good
a = f( 1, 2 ) + g( 3, 4 ) # bad
```

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.

More: PEP 8 and https://pep8.org/

General Good Practices

Naming

Use `snake_case` for variables/functions; `CamelCase` for classes; `CAPS_CASE` for constants.

Decomposition and Logic

Same as 106A/B/X. Simple is better than complex!

Automated Code Style Checking

Use PEP8 Online for mechanical violations (naming, spacing) and more advanced suggestions.

Use `pycodestyle` as a command line tool. Install with `pip install pycodestyle`.

More: PEP 8 and https://pep8.org/

Remember the Zen of Python

First-Class Functions

Functions are Objects!

First-Class Functions

```
def echo(arg):
    return arg

type(echo) # => <class 'function'>
id(echo) # => 4393732128
print(echo) # => <function echo at 0x105e30820>

foo = echo
type(foo) # => <class 'function'>
id(foo) # => 4393732128
print(foo) # => <function echo at 0x105e30820>

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

Questions

What can you do with function objects?

What attributes does a function object possess?

Can I pass a function as a parameter to another function?

Can a function return another function?

How can I modify a function object?

We must go deeper. (Lab)

Summary

Summary

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 (?!)