short answer,concept
input output question
programming coding: practicing problem

Lecture 3: Advanced Functions

MPCS 51042-2: Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Oct 21, 2019

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Local and Remote-Tracking Branches

- A branch is a pointer to a particular commit
- A local branch points to a commit on your local repo.
  - Changed by local operations like `git commit`
  - Simply named, such as “master”

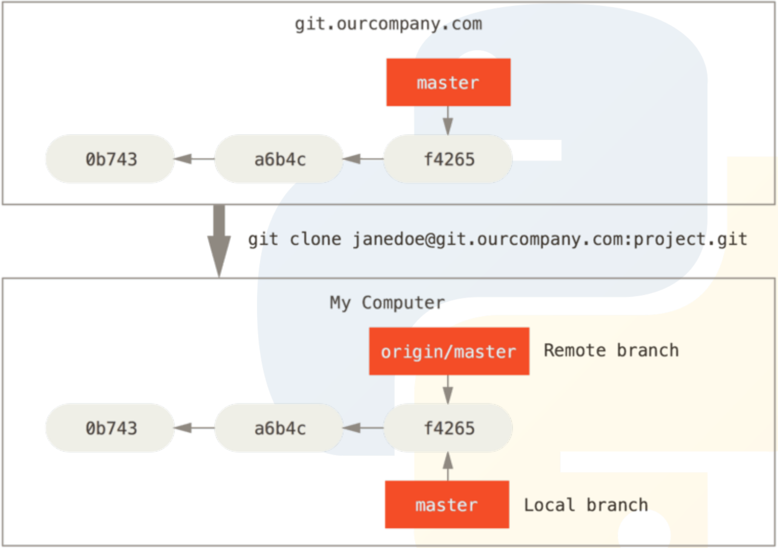
gitlab

- A remote-tracking branch points to a commit on a remote repo.
  - Named with both remote and branch name, such as “origin/master”

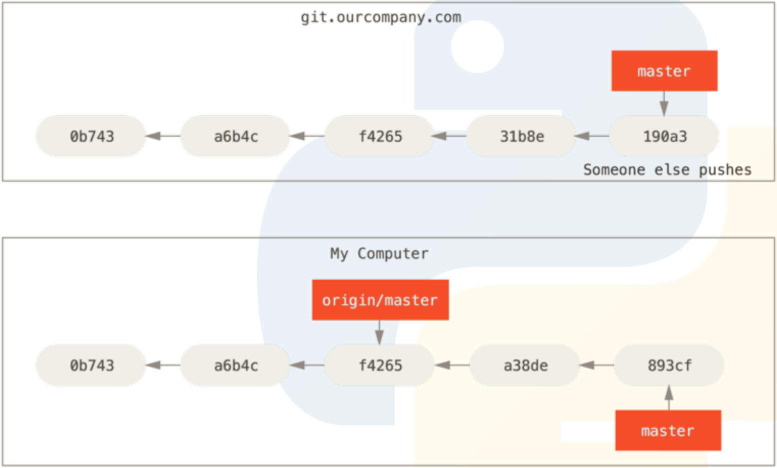
Updating Remote-Tracking Branches

- Your local repo's remote-tracking branches are not automatically updated when the remote itself changes.
- Use `git fetch` to update your remote-tracking branches:
  - `git fetch <remote_name>`: update all remote-tracking branches that point to a given remote
  - `git fetch --all`: update all remote-tracking branches from all remotes
- this does not change your local branches or your working tree.

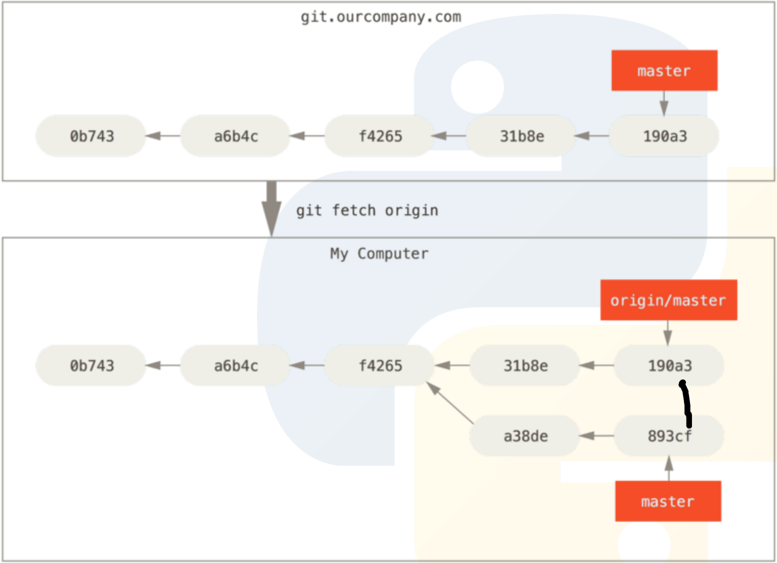
Example: A freshly-cloned repo



Example: Someone else pushes commits to remote



Example: Fetching new commits from remote



Merging Commits from Remote

Two ways to get changes from remote-tracking branch into local branch.

- Fetch and merge:

```
git fetch origin
git merge origin/master
```

- Pull:

```
git pull origin master
```

Can also name other remotes or branches:

```
git pull upstream master
```

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

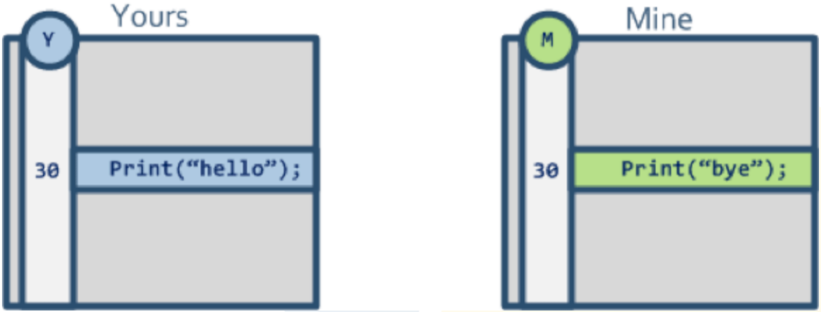
Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Source

- “Three-way Merging: A Look Under the Hood”: `http://blog.plasticscm.com/2016/02/three-way-merging-look-under-hood.html`

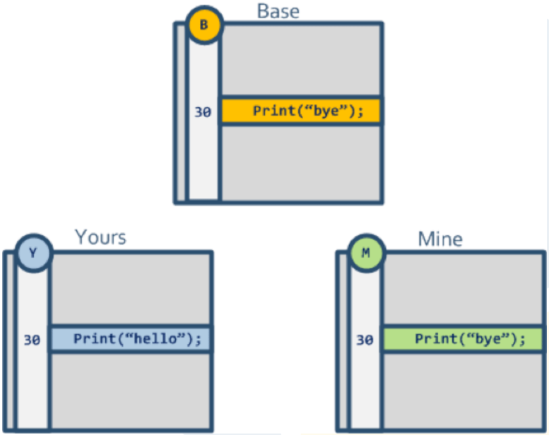
A Two-Way Merge (Hypothetical)



- You and me have different code on line 30
- Not enough info to know whose to keep

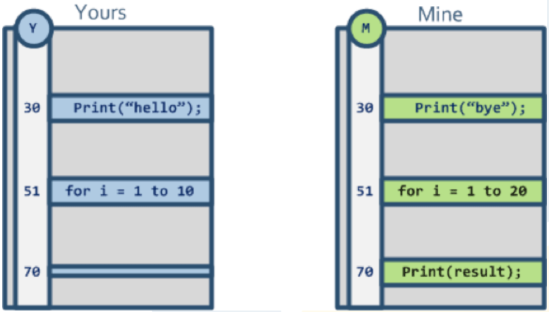
A Three-Way Merge (Actual)

yours is
more recent,
so keep this



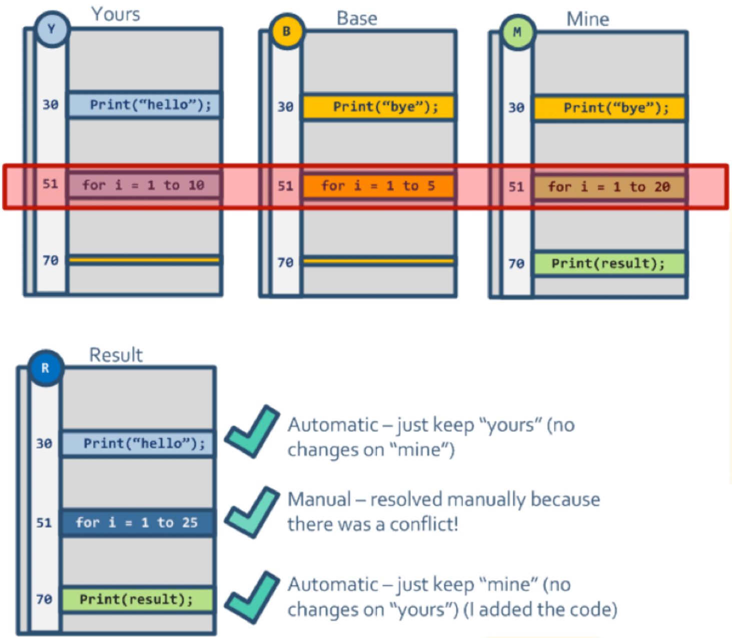
- Compares yours and mine to common ancestor
- Now it's clear that we'll keep your line 30

Multiple Lines in a Three-Way Merge



- Changes are compared and resolved on a line-by-line basis

Automatic and Manual Merging



Resolving Merge Conflicts

- Simplest way is to look in the file itself
- Lines with merge conflict will be marked like this:

```
If you have questions, please
<<<<<<< HEAD
open an issue
=======
ask your question in IRC.
>>>>>>> branch-a
```

edit manually

- For more details:
  - https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line
  - https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Pulling Upstream Changes for Homework

1. Add upstream

```
git remote add upstream git@mit.cs.uchicago.edu:mpcs51042-aut-19/mpcs51042-2-aut-19.git
```

2. Pull upstream changes

```
git pull upstream master
```

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Anonymous Functions

- A `lambda` expression creates and returns a function object.
  - A `def` statement creates a function and assigns it to a name.
  - A `lambda` expression is often not assigned to a name (for example, as an inlined argument to another function).
- A `lambda`'s body is a single expression, not a block of statements.

Example: Custom Sorting

- The `sorted()` function takes an optional argument for custom sorting.
- This argument, `key`, is a function used to obtain the comparison key for each element.

```
>>> ron = dict(skin_color='brown', hair_color='black',
...            eye_color='brown', fav_color='azure')

>>> sorted(ron.items())
[('eye_color', 'brown'), ('fav_color', 'azure'),
('hair_color', 'black'),  ('skin_color', 'brown')]

>>> sorted(ron.items(), key=lambda x: x[1])
[('fav_color', 'azure'), ('hair_color', 'black'),
('skin_color', 'brown'), ('eye_color', 'brown')]
```

sorted by keys

sorted by values

More Lambda Examples

- Sort strings using true alphabetical ordering, rather than "ASCII-betical" ordering
- Show how to make a dict-of-dict-of-lists using defaultdict

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Functional Programming

Functional programming tools apply an operation to every item in an iterable.

- Two built-in functions:
  - `map(function, iterable, ...)`: Applies a general function to every item in the passed iterable. Returns a new iterator for results.
  - `filter(function, iterable)`: Returns all elements for which function is `True`. Result is a new iterator.
- Also provided in `functools` module:
  - `functools.reduce(function, iterable[, initializer])`: Apply function cumulatively and return a single value as the result. Optionally specify the initial value.
- Others in `itertools` module: filterfalse, dropwhile, takewhile, etc.
- Standard operators are expressed as functions in the `operator` module.

Using Map

Works with built-ins:

```
>>> list(map(math.ceil, [1.1, 2.5, 3.01]))
[2, 3, 4]

>>> set(map(str.upper, {'apple', 'Banana', 'CHErry'}))
{'BANANA', 'CHERRY', 'APPLE'}
```

The operator module provides functions for built-in operators

```
>>> from operator import neg
>>> list(map(neg, [1.1, 2.5, 3.01]))
[-1.1, -2.5, -3.01]
```

Using Map, cont.

Great place to use lambdas:

```
>>> tuple(map(lambda x: x+10, [1, 3, 5]))
(11, 13, 15)
```

map function generate iterable object

```
>>> f = lambda s: str.capitalize(s) + " are great!"
>>> set(map(f, {'apples', 'Bananas', 'CHErries'}))
{'Apples are great!', 'Cherries are great!',
'Bananas are great!'}
```

Using Map with Multiple Iterables

Map can use an N-argument function to handle N iterables:

- pow takes 2 arguments; map gives it 2 iterators:

```
>>> list(map(pow, [2, 4, 8], [6, 3, 2]))
[64, 64, 64]
```

- The operator module has functions for binary operators:

```
>>> from operator import mul
>>> list(map(mul, [2, 4, 8], [6, 3, 2]))
[12, 12, 16]
```

- max takes an arbitrary number of args:

```
>>> list(map(max, [1, 7, 24], [5, 1, 0], [2, -20, 100]))
[5, 7, 100]
```

Too Much Work in a Map?

Is this clearer than a for-loop or generator?

```
>>> d = {'dog': 'mammal', 'shark': 'fish',
...       'duck': 'dinosaur'}
>>> joiner = lambda x: str.join(' is a ', x)
>>> list(map(joiner, d.items()))
```

-

Filter

Filter returns all elements for which the test function is true:

```
>>> list(filter(str.isalpha, ["can't", "abc", "2nd"]))
['abc']

>>> list(filter(lambda x: x % 2 == 0, [22, 1, 0, 4.1, 15]))
[22, 0]
```

Reduce

Reduce takes a 2-argument function and applies it cumulatively to the elements in an iterable:

```
>>> from functools import reduce
>>> from operator import add, mul
>>> reduce(mul, [1, 2, 3, 4])
24
```

Takes an optional initializer:

```
>>> reduce(add, [1, 2, 3, 4], 10)
20
```

A user-defined function must take two args:

```
>>> reduce(lambda x, y: -(x+y), [1, 2, 3, 4])
-4
```

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

List Comprehensions

A list comprehension will:

- Apply an arbitrary expression (not function) to an iterable.
- Create a list (not an iterator) of results.

Comprehensions can also create dictionaries and sets.

For-Loop vs. Map vs. Comprehension

All of these produce the values [0, 1, 4, 9, 16].

- All of them take the same iterable.
- Map applies a function and returns an iterator.
- Comprehension applies an expression and returns a list.

[expression
for target 1 in iterable1 if condition
for target 2 in iterable 2 if .....]

```
# For loop
L = []
for i in range(5):
    L.append(i ** 2)

# Map (an iterator, not list)
I = map(lambda i: i ** 2, range(5))

# Comprehension
L = [i ** 2 for i in range(5)]
```

Nesting and Conditionals in Comprehensions

cannot use lambda
function here

- Comprehensions can be arbitrarily nested.
  - The first for-loop is outermost.
  - Subsequent for-loops are nested inwards.

```
L = [x+" "+y for x in ('one', 'two') for y in ('fish', 'car')]
#  Returns ['one fish', 'one car', 'two fish', 'two car']
```

- Comprehensions can use conditions to “filter” the iterable:

```
L = [s.upper() for s in ('cat', 1.23, 'dOG', [])
     if isinstance(s, str)]
# Returns ['CAT', 'DOG']
```

Set and Dictionary Comprehensions

- Set comprehensions have the generator syntax:

```
{f(x) for x in iterbl if P(x)}
```

- Dictionary comprehensions have the general syntax:
  - `iterbl` should be an iterable of `(key, value)` pairs.
  - For example, `zip(keys, vals)` produces a suitable result.

```
{f(k): g(v) for (k,v) in iterbl if P(k,v)}
```

Example: Cartesian Product

Write two functions that use a for loop and a comprehension to get the Cartesian Product of two iterables:

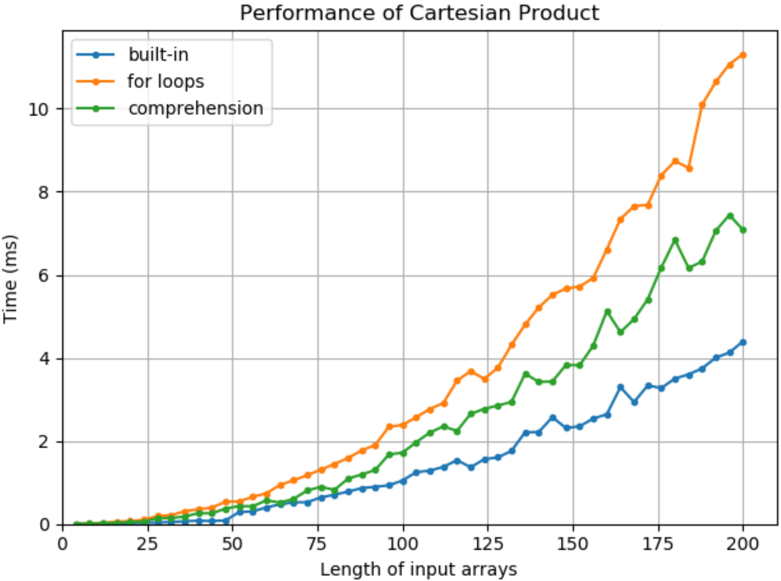
```
>>> compr_cart_product([1, 2, 3], ['A', 'B', 'C'])
[[1, 'A'], [1, 'B'], [1, 'C'], [2, 'A'], [2, 'B'], [2, 'C'],
[3, 'A'], [3, 'B'], [3, 'C']]
```

Example: Cartesian Product, cont.

```
def for_loop_cart_product(list1, list2):
    results = []
    for i in list1:
        for j in list2:
            results.append([i, j])
    return results

def compr_cart_product(list1, list2):
    return [[i, j] for i in list1 for j in list2]
```

Performance of Cartesian Products



More Comprehension Examples

- Split a string of text into a list of sentences.
- Split a string of text into a nested list of sentences and words
-

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Table of Contents

Git Remotes and Merging

Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
How Merging Works
Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

Anonymous Functions (lambdas)
Map, Reduce, and Filter
List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

Generator Functions
Generator Statements

Variable Scopes (Lutz Ch. 17)

Assignment and Reference Rules
Global and Local Scopes
Nonlocal Scope and Closures
Function Wrappers
Function Attributes

Generator Functions

- Generator functions provide another way to retain state.
  - They suspend their state between multiple calls.
  - They are compiled into generator objects, which are a kind of iterator.
- They are written to yield a value to the caller, then resume execution.
- Returning or exiting the function terminates the execution.

Generating Fibonacci Numbers

- This yields Fibonacci numbers one at a time.
- Each iteration stops and resumes at `yield`.
- The iteration ends when the function exits.

```
def fib(end):
    last = 0
    curr = 1
    for i in range(end):
        yield curr
        nxt = curr + last
        last = curr
        curr = nxt

for i in fib(5):
    print(i, end=": ")  # prints 1: 2: 3: 5: 8:
```

Non-terminating Fibonaccis

Sometimes it is useful to make a non-terminating generator:

```
def fib():
    last = 0
    curr = 1
    while True:                          # An infinite loop
        yield curr                      stop and returns curr
        nxt = curr + last
        last = curr
        curr = nxt

f = fib()
for i in range(5):
    print(next(f), end=": ")     # Calling next() manually yields

print("\nLet's take a break...")

for i in range(5):
    print(next(f), end=": ")     # Begin yielding again
```

Other Ways to Work with Iterators

- The `itertools` module has efficient functions for iterables.
- E.g., `islice` returns selected elements from an iterable (like slicing).

```
def fib():
    last = 0
    curr = 1
    while True:
        yield curr
        nxt = curr + last
        last = curr
        curr = nxt

from itertools import islice
for i in islice(fib(), 2, 6):    # prints 2: 3: 5: 8:
    print(i, end=": ")
```

Table of Contents

Git Remotes and Merging

- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)

- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Generator Statements

- Generator statements produce generator objects using syntax similar to comprehensions.
- Here, both `x` and `y` are generator objects that yield [-5, 10, 15, -20]
- The generator expression is more concise here. However, for more complicated cases, a generator function can retain much more state information and have more internal logic (recall earlier examples in lecture).

```python
def mygen(L):
    for i in L:
        yield i * 5

x = mygen([-1, 2, 3, -4])

y = (i * 5 for i in [-1, 2, 3, -4])
```

Generator Statements vs. Comprehensions

- Biggest difference: generator expressions construct them one-at-a-time, but comprehensions construct results all at once.
- This often introduces space/time trade-off when consuming entire iterable:
  - Generator expressions often use less memory than the equivalent comprehension.
  - Comprehensions often run faster over all iterations.
- If the entire iterable will not be necessarily be consumed, generators can also be a better choice.

don't know how much data to use

Generator Statements vs. Map

Clarity and conciseness depends on the situation:

```
L = [-1, 2, 3, -4]

# Generator expression may be clearer.
x = map(lambda i: i*10, L)
y = (i*10 for i in L)

# Map may be clearer.
x = map(abs, L)
y = (abs(i) for i in L)
```

Generator Statements vs. Filter

- Conditionals are allowed in generator expressions (like in comprehensions)
- This allows generator expressions to emulate `filter`

```
L = ['at', 'cat', 'scat']

x = filter(lambda s: len(s) > 2, L)

y = (s for s in L if len(s) > 2)
```

Generator Statements vs. Map/Filter

Generator expressions are often more concise than combining `map` and `filter`

```
L = ['at', 'cat', 'scat']

x = map(str.upper, filter(lambda s: len(s) > 2, L))
print(list(x))

y = (s.upper() for s in L if len(s) > 2)
print(list(y))
```

generator: ()
comprehension:[]

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Table of Contents

Git Remotes and Merging

Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
How Merging Works
Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

Anonymous Functions (lambdas)
Map, Reduce, and Filter
List, Set, and Dictionary Comprehensions

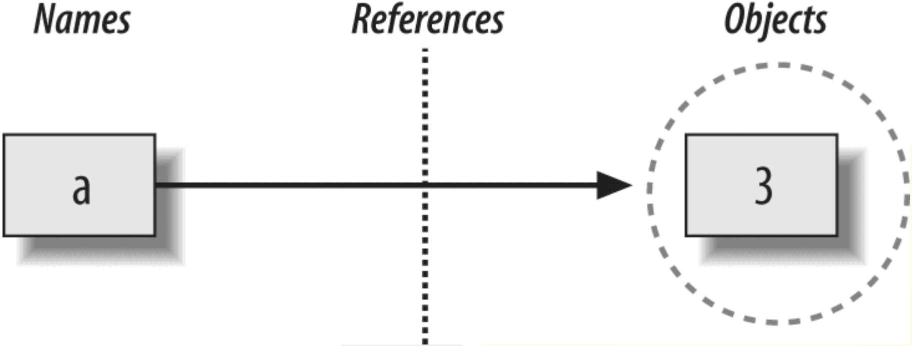
Generators (Lutz Ch. 19 and 20)

Generator Functions
Generator Statements

Variable Scopes (Lutz Ch. 17)

Assignment and Reference Rules
Global and Local Scopes
Nonlocal Scope and Closures
Function Wrappers
Function Attributes

Names, References, and Objects



A name (the variable) refers to an object (the data itself).

Assignment vs. Reference

An assignment does one of the following:

- Creates a new name that refers to a new/existing object.
- Makes an existing name refer to a different object.

```
L = [1, 2, 3]     # Creates a new name, L, and new object
L = [4, 5, 6]     # L points to a different object
```

A reference looks-up a name and retrieves the object.

```
print(L[0])       # Finds the object that L[0] points to
```

Question: Is in-place modification an assignment or a reference?

```
L.append(99)
```

reference: look up
the object and
modify

The 3 Scopes: Global, Enclosing, and Local

- By default, a variable's scope is determined by where it was first assigned.
- Relative to a given function definition, there are three scopes:
  - Global variable: assigned inside the surrounding module. There are no variables that span multiple modules.
  - Nonlocal variable: assigned in a surrounding functions. This extends to arbitrarily-many nested functions.
  - Local variable: assigned inside the given function.
- Respective to `inner_func`, these `X` are all different instances:

```
X = 'global_to_inner_func'
def outer_func():
    X = 'nonlocal_to_inner_func'
    def inner_func():
        X = 'local_to_inner_func'
```

all Xs are different

Name Resolution

For variable references, Python searches in this order:

- The local scope
- The enclosing scope
- The global scope
- Built-in names

For variable assignements:

- If the variable is unqualified, a local variable is always created or changed.
- If the variable is qualified with the `global` or `nonlocal` keywords, then the global/nonlocal variable is changed.

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Global Reference

In `adder()`, when `x` is referenced, it is found in the global scope.

```python
x = 8                  # Creates a global variable

def adder(y):
    return x + y       # Finds x in global scope

print(adder(2))        # Prints 10
```

Question: What is the scope of `y`?

local

Local Assignment and Reference

Now there are multiple instances of `x`:

- When `x` is referenced in `adder()`, the local variable is found.
- When `x` is referenced in global scope, the global variable is found.

```python
x = 99

def adder(y):
    x = 1           # Creates local instance of x
    return x + y    # Finds x in local scope

print(adder(2))     # Prints 3
print(x)            # Prints 99, since global x is unchanged
```

Global Assignment: The global Namespace Declaration

If we want to assign a variable in the global scope, we use the `global` namespace declaration.

```
x = 99

def adder(y):
    global x
    x = 1           # Re-assigns global x
    return x + y    # Finds x in in global scope

print(adder(2))     # Prints 3
print(x)            # Prints 1, since global x was reassigned
```

Table of Contents

Git Remotes and Merging

Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
How Merging Works
Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

Anonymous Functions (lambdas)
Map, Reduce, and Filter
List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

Generator Functions
Generator Statements

Variable Scopes (Lutz Ch. 17)

Assignment and Reference Rules
Global and Local Scopes
Nonlocal Scope and Closures
Function Wrappers
Function Attributes

Enclosing (Nonlocal) References

When `x` is referenced inside `inner()`, it is found in the enclosing scope.

```python
x = 99              # x is global

def outer():
    x = 1           # x is local to outer()
    def inner():
        print(x)    # Gets x from enclosing scope
    inner()

outer()             # Prints "1"
```

Nonlocal Arguments in Outer Function

Arguments to the outer function are nonlocal to the inner.

```
def outer(base):
    exp = 2                      # Both base and exp are local
    def inner():
        print(base ** exp)   # Both base and exp are nonlocal
    inner()

outer(5)                         # Prints "25"
```

5 is nonlocal

Closures (or Factory Functions)

- A function object retains values in its nonlocal scope.
- You can use this to generate functions that retain state.

```
def outer(base):
    def inner(exp):
        print(base ** exp) # Base is nonlocal, exp is local
    return inner
five_pow = outer(base=5)    # base=5 is in five_pow's closure
five_pow(exp=2)             # Prints "25"
five_pow(exp=3)             # Prints "125"
```

when called outer, create new instance of inner function

then 5 here is a five_pow's closure

Independent States

Each function object gets a separate scope, even when created by the same factory.

```
def outer(base):
    def inner(exp):
        print(base ** exp)  # Base is nonlocal, exp is local
    return inner

five_pow = outer(base=5)    # base=5 is in five_pow's closure
five_pow(exp=2)             # Prints "25"

ten_pow = outer(base=10)    # base=10 is in new closure
ten_pow(exp=2)              # Prints "100"

five_pow(exp=3)             # Still has base=5 in closure
```

Nonlocal Assignments: The `nonlocal` Declaration

- Previously, we were referencing a nonlocal variable.
- To reassign a nonlocal variable, we must declare it as `nonlocal`.

```
def outer(base):
    call_count = 0
    def inner(exp):
        nonlocal call_count
        call_count += 1
        print("ans: {}, call count: {}".format(
                base**exp, call_count))
    return inner

five_pow = outer(base=5)
five_pow(exp=2)                # ans: 25, call count: 1

ten_pow = outer(base=10)
ten_pow(exp=2)                 # ans: 100, call count: 1

five_pow(exp=3)                # ans: 125, call count: 2
```

Table of Contents

Git Remotes and Merging

Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
How Merging Works
Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)

Anonymous Functions (lambdas)
Map, Reduce, and Filter
List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)

Generator Functions
Generator Statements

Variable Scopes (Lutz Ch. 17)

Assignment and Reference Rules
Global and Local Scopes
Nonlocal Scope and Closures
Function Wrappers
Function Attributes

Function Wrappers

A function wrapper takes one function and returns another:

```
def counter(func):                                  # Pass a callable
    call_count = 0
    def inner(*args, **kwargs):
        nonlocal call_count
        call_count += 1
        print("call count: {}".format(call_count))
        return func(*args, **kwargs)                # Returns result
    return inner

pow_count = counter(pow)
x = pow_count(5, 2)          # Returns 25,  prints "call count: 1"
x = pow_count(5, 3)          # Returns 125, prints "call count: 2"

min_count = counter(min)
y = min_count([3, 5, 1, 9]) # Returns 1, prints "call count: 1"
y = min_count(3, 5, 1, 9)   # Returns 1, prints "call count: 2"
```

Table of Contents

Git Remotes and Merging
- Local and Remote-Tracking Branches (Chacon & Straub Ch. 3.5)
- How Merging Works
- Pulling Upstream Changes for Homework

Functional Programming (Lutz Ch. 19)
- Anonymous Functions (lambdas)
- Map, Reduce, and Filter
- List, Set, and Dictionary Comprehensions

Generators (Lutz Ch. 19 and 20)
- Generator Functions
- Generator Statements

Variable Scopes (Lutz Ch. 17)
- Assignment and Reference Rules
- Global and Local Scopes
- Nonlocal Scope and Closures
- Function Wrappers
- Function Attributes

Function Attributes

- Function attributes are another way to maintain state.
- The attribute is nonlocal in the scope of `inner`.
- The attribute is accessible to the caller as an instance attribute.

```
def counter(func):                                  # Pass a callable
    def inner(*args, **kwargs):
        inner.call_count += 1
        print("call count: {}".format(inner.call_count))
        return func(*args, **kwargs)                # Returns result
    inner.call_count = 0
    return inner

pow_count = counter(pow)
x = pow_count(5, 2)          # Returns 25,  prints "call count: 1"
x = pow_count(5, 3)          # Returns 125, prints "call count: 2"
c = pow_count.call_count     # Accessible to caller's scope
print(c)
```