CIS 1100

Functional Programming in Python

Python
Fall 2024
University of Pennsylvania

Learning Objectives

- Use functions as *first-class values* that can be passed around and manipulated in a program
- Use *higher-order functions* that take other functions as inputs in order to develop another approach to the common patterns of *mapping*, *filtering*, and *aggregating*.
- Use *lambdas* to define short, "single-use" functions with an abbreviated syntax.

CIS 1100

Functions as Objects

Python
Fall 2024
University of Pennsylvania

Functions are Objects

Function objects are defined using the function syntax—exactly how we've defined them before.

```
def my_zip(a, b):
    """Create a list of pairs of elements from two sequences.

    Arguments:
    a -- The first sequence
    b -- The second sequence

    Returns:
    a list containing pairs of elements from the input sequences.
    """
    zipped = []
    smaller_len = min(len(a), len(b))
    for i in range(smaller_len):
        zipped.append((a[i], b[i]))
    return zipped
```

Function Names Are Identifiers

We've always used functions as things that can be *defined* and then *called*

```
>>> my_zip("haha", "lololol")
[('h', 'l'), ('a', 'o'), ('h', 'l'), ('a', 'o')]
```

But functions are just regular data.

```
>>> my_zip
<function my_zip at 0x1057101f0>
```

And regular data can be saved inside of other variables...

```
>>> f = my_zip
>>> f
<function my_zip at 0x1057101f0>
>>> f("123", "abc")
[('1', 'a'), ('2', 'b'), ('3', 'c')]
```

Objects Have Attributes

```
>>> f.__name__
'my_zip'
>>> f.__doc__
'Create a list of pairs of elements from two sequences.\n\n    Arguments:\n...'
>>> f.__code__.co_argcount
2
```

Disclaimer: no need to memorize these!

Functions Can Be Saved in Lists

```
analyses = [max, min, sum] # built-in functions
my_numbers = [0.96042771, 0.1607802, 0.006373713]

for analysis in analyses:
    print(f"{analysis.__name__} of my_numbers is {analysis(my_numbers)}")
```

```
max of my_numbers is 0.96042771
min of my_numbers is 0.006373713
sum of my_numbers is 1.127581623
```

Towards Higher Order Functions

These features from the previous slides are not going to be vital for you as beginner programs. Instead, they reveal a new way of thinking about functions as data. We will now develop this further.

CIS 1100

Higher Order Functions

Python
Fall 2024
University of Pennsylvania

Higher Order Functions

Higher Order Functions are functions that take *other functions* as input.

This is possible since functions are just objects, and objects can be passed around in programs.

`min()` as Higher Order

Previously, `min()` could be used to find the smallest value in some sequence:

```
>>> min([4, 15, 100, -20, 95])
-20
```

`min()` as Higher Order

What happens if we try to use `min()` on sequences of strings?

```
>>> names = ["Ivan", "Margaret", "Oly", "Sylvia"]
>>> min(names)
"Ivan"
```

By default, `min()` orders strings by comparing them alphabetically.

`min()` as Higher Order

What if we wanted the person with the shortest name?

- The *key* for comparison is therefore the length
- We have a function, `len`, that computes the length of a string

```
>>> names = ["Ivan", "Margaret", "Oly", "Sylvia"]
>>> min(names, key=len)
"Oly"
```

We set the keyword argument `key` to take the value `len`, which compares pairs of elements `a` and `b` by `len(a) < len(b)` instead of `a < b`

Higher Order Functions: Why?

Many of our problems are just specific instances of common & general patterns

- **Filtering:** Removing all elements from a sequence that do not meet a particular condition
- **Mapping:** Applying some transformation to all elements of a sequence
- **Aggregating:** Performing some aggregate calculation based on the elements of a sequence

CIS 1100

Filtering

Python
Fall 2024
University of Pennsylvania

Filter Values Out of a Sequence

```
new_list = []                           # [] is a list with no contents
for variable in sequence:               # For each value in the source sequence,
  if condition(variable):               # if that value meets some condition
    new_list.append(<expression>)       # add that value to the end of the new list.
```

We can rewrite the loop (above) into the comprehension (below)

```
new_list = [<expression> for variable in sequence if condition(variable)]
```

Recall: Getting Non-Zero Exam Scores

With a loop...

```
exam_scores = [100, 0, 89, 93, 78, 67, 0]
non_zeroes = []                  # [] is a list with no contents
for score in exam_scores:        # For each score from the list,
  if score > 0:                  # if that score is not zero,
    non_zeroes.append(score)     # add that score to the end of the new list.
```

...or a comprehension:

```
exam_scores = [100, 0, 89, 93, 78, 67, 0]
non_zeroes = [score for score in exam_scores if score > 0]
print(non_zeroes)
```

🖨️👇

```
[100, 89, 93, 78, 67]
```

`filter`

`filter` is a higher order function that takes in a function and sequence and returns a new sequence containing only those elements for which the provided function evaluates to `True`.

`filter(f, seq)` is equivalent to:

```
[elem for elem in seq if f(elem)]
```

`filter`

Rules about `filter`:

- the function being passed in must be one that can be called with a single argument.
- the function being passed in must be one that produces a boolean value.
- the value returned by `filter` is not actually a list, but can be converted to one using `list()`

Recall: Getting Non-Zero Exam Scores

Define a function...

```
def is_positive(n):
    return n > 0
```

...and use it in `filter()`:

```
>>> result = filter(is_positive, [100, 0, 89, 93, 78, 67, 0])
>>> result
<filter object at 0x105711a90>
>>> l = list(result)
>>> l
[100, 89, 93, 78, 67]
```

Recall: Checking Capitalization

```
names = ["haRry", "Adi", "molly", "jared", "cEDRIc", "Sukya", "TraviS"]
proper_caps = []                  # [] is a list with no contents
for name in names:                # For each name from the list,
  if name.istitle():              # if that name is in "title case"
    proper_caps.append(name)      # add that name to the end of the new list.
print(proper_caps)
```

🖨️👇

```
["Adi", "Sukya"]
```

Recall: Checking Capitalization

Could try `filter(istitle, names)`...

```
>>> names = ["haRry", "Adi", "molly", "jared", "cEDRIc", "Sukya", "TraviS"]
>>> result = filter(istitle, names)
NameError: name 'istitle' is not defined
```

`istitle()` is actually a method belonging to the `str` class. It's not a free-floating function.

Recall: Checking Capitalization

Could try `filter(str.istitle, names)` instead

```
>>> names = ["haRry", "Adi", "molly", "jared", "cEDRIc", "Sukya", "TraviS"]
>>> result = filter(str.istitle, names)
>>> list(result)
['Adi', 'Sukya']
```

✅

CIS 1100

Mapping

Python
Fall 2024
University of Pennsylvania

Transforming Values in a Sequence

Curving exam scores using comprehensions...

```
exam_scores = [92, 99, 100, 98.5]
curved_scores = [score + 10 for score in exam_scores]
```

...or just loops:

```
curved_scores = []
exam_scores = [92, 99, 100, 98.5]
for score in exam_scores:
  curved_scores.append(score + 10)
```

`map`

`map` is a higher order function that takes in a function and sequence and returns a new sequence containing elements of the input sequence after having `f` applied to them.

`map(f, seq)` is equivalent to:

```
[f(elem) for elem in seq]
```

`map`

Rules about `map`:

- the function being passed in must be one that can be called with a single argument.
- the function being passed in must can return a value of any type.
- the value returned by `map` is not actually a list, but can be converted to one using `list()`

Recall: Curving Exam Scores

Define a function...

```
def add_ten(n):
    return n + 10
```

...and use it in `map()`:

```
>>> result = map(add_ten, [100, 0, 89, 93, 78, 67, 0])
>>> result
<map object at 0x105711a90>
>>> l = list(result)
>>> l
[110, 10, 99, 103, 88, 77, 10]
```

Mapping Using Built-Ins

We could transform a list of strings into a list of their lengths using `len`:

```
>>> names = ["hss", "tQm", "aditya", "Sukya"]
>>> lengths = list(map(len, names))
>>> lengths
[3, 3, 6, 5]
```

Mapping Using Built-Ins

We could transform a list of strings into a list of lowercase strings using `str.lower`:

```
>>> names = ["hss", "tQm", "aditya", "Sukya"]
>>> lowercase = list(map(str.lower, names))
>>> lowercase
['hss', 'tqm', 'aditya', 'sukya']
```

CIS 1100

Aggregating

Python
Fall 2024
University of Pennsylvania

`reduce`

Aggregation is done using a slightly trickier HOF called `reduce` imported from `functools`.

`reduce` is a function that takes in an *accumulator function* and a sequence and repeatedly accumulates elements from the sequence using the accumulator function.

`reduce(f, seq)` is roughly equivalent to:

```
result = seq[0]
for elem in seq[1:]:
    result = f(result, elem)
```

`sum` as `reduce`

If we define:

```
def add_two_numbers(a, b):
    return a + b
```

Then `reduce(add_two_numbers, [3, 4, 5, 6])` evaluates to:

```
(((3 + 4) + 5) + 6)
```

`reduce`

Rules about `reduce`:

- the function being passed in must be one that can be called with two arguments.
  - the first argument represents the accumulator value
  - the second argument represents the next element from the sequence
- the function can return a value of any type
- the value returned can be of any type, depends on the input function
- if the iterable is empty, then the `reduce` will cause an error
  unless an optional third initializer argument is passed in.

Initializer in `reduce`

`reduce` can take an optional third argument
specifying which value to start the accumulator at:

`reduce(f, seq, init)` is roughly equivalent to:

```
result = init
for elem in seq:
    result = f(result, elem)
```

Counting Even Values Using `reduce`

```
def increment_if_even(acc, elem):
    if elem % 2 == 0:
        return acc + 1
    else:
        return acc
```

```
>>> nums = [13, 45, 18, 10, 20, 31]
>>> reduce(increment_if_even, nums, 0)
3
```

CIS 1100

Lambdas

Python
Fall 2024
University of Pennsylvania

Function Definitions

So far, functions are defined with:

- names,
- input variables,
- and bodies

But we know that we can assign arbitrary identifiers to functions by saving them in variables. Maybe the name thing isn't that important?

Lambdas

Lambdas (or anonymous functions) are functions that are defined without names.

```
lambda <parameter_list> : expression
```

would become

```
def no_name(<parameter_list>):
    return expression
```

Advantages of Lambdas

A lot of higher order functions take *simple* functions as inputs.

```
def add_two_numbers(a, b):
    return a + b
reduce(add_two_numbers, l, 0)
```

could be replaced with:

```
reduce(lambda a, b : a + b, l, 0)
```

Advantages of Lambdas

A lot of higher order functions take *simple* functions as inputs.

```
def is_positive(c):
    return c > 0
filter(is_positive, l)
```

could be replaced with:

```
filter(lambda c : c > 0, l)
```

Disadvantages of Lambdas

- Reduce readability
- Can only express simple one-liners
- Can't be "saved" for later without saving in a variable
  - this is bad style to do
  - just make a `def` statement instead

`max()` as Higher Order

Previously, `max()` could be used to find the largest value in some sequence:

```
>>> max([4, 15, 100, -20, 95])
100
```

`max()` as Higher Order

What happens if we try to use `max()` on sequences of tuples?

```
>>> records = [("Ivan", 394), ("Peter", 832), ("Naomi", 398), ("Sylvia", 213)]
>>> max(records)
("Sylvia", 213)
```

By default, `max()` orders tuples by comparing their first values, and so `("Sylvia", 213)` is greatest based on alphabetical ordering.

`max()` as Higher Order

What if we wanted the person with the highest score?

```
>>> records = [("Ivan", 394), ("Peter", 832), ("Naomi", 398), ("Sylvia", 213)]
>>> max(records, key=lambda t: t[1])
("Peter", 832)
```

We can provide an anonymous function `lambda t: t[1]`

to tell `max()` to compare our based on their second elements.