Review, Lambdas, Comprehensions

6.100 LECTURE 15

SPRING 2026

Announcements

- **Pset 4 checkoffs** throughout this week
  - in second half of term, checkoff discussions will be more challenging
  - low-stakes for your grade, chance to engage in deep discussion
  - grading expectations
    - check-pluses are uncommon, means you got everything
    - checks are fine, means you learned something
    - check-minus means you need to think more carefully
- **Exam 2 next Monday 4/6 in Walker**
  - covers material up through Week 7 before break
  - practice exam and additional problems to be posted after class
  - also review Exam 1, lectures, recitations, exercises, psets

Subject Review

Up through Exam 1

- objects, variables
- functions and model of environment frames
- iteration (over **`str`**, **`range`**, **`list`**), exhaustive enumeration
- stochastic simulation, sampling
- distributions, Central Limit Theorem

Up through Exam 2

- statistical confidence, significance
- optimization, linear regression, training/validation
- bisection search
- more iterables (over **`tuple`**, **`set`**, **`dict`**)
- graph modeling, shortest paths, BFS, Dijkstra’s
- DFS, recursion

Up through Exam 3

- Python conveniences
  - lambda functions
  - conditional expressions
  - comprehensions
- more recursion
  - discrete/combinatorial optimization
  - dynamic programming
- object-oriented programming
  - classes
  - inheritance
- defensive programming
  - exceptions
  - assertions, testing

Lambda Functions

Function objects vs other types

- A function definition creates a function object and then points a variable name to it

  -
    ```
    def func(x, y):
        statement...
    ```
- For all other types, we use ***var = expr*** assignment notation
  - often, ***expr*** can be a literal object
  - numbers, strings, lists, tuples, dictionaries, etc. all have literal representations
- Is there an equivalent literal representation for functions?

Lambda functions

- A way to create function objects not assigned a name
  - “anonymous” functions
- **Syntax:** **`lambda x, y:`** ***`expr`***
  - “lambda” means “create a function object”
  - parameter list just like in regular functions, but no parens needed
  - only a **single expression** may follow the colon, that is what this function evaluates and returns
- Equivalent forms
  - ***`func`*** **`= lambda x, y:`** ***`expr`***

  -
    ```
    def func(x, y):
        return expr
    ```
  - the regular syntax (latter) is preferred if you’re going to name a function

Typical uses of lambda functions

- short but custom specifications of behavior
  - e.g., generalizing bisection search for square roots to any monotonically increasing function
    - cube root: **`bisection_search(num, lambda x: x**3)`**
    - logarithm: **`bisection_search(num, lambda x: 2**x)`**
- key functions for **`min()`**, **`max()`**, **`sorted()`**
  - given a collection (iterable) of elements, chose a custom specification of how the elements should be compared
  - e.g., suppose the elements are **`str`** names
    - sort by length: **`sorted(names, key=len)`**
    - sort by last letter: **`sorted(names, key=lambda x: x[-1])`**

Limitations of lambda functions

- Not all Python functions can be written as lambdas
- **`lambda`** is meant for simple transformations of input
  - hence no explicit **`return`**
- **`lambda`** cannot handle arbitrary sequences of statements
- E.g., mutate a sequence of numbers to keep only positives

  -
    ```
    def func(seq):
        for i in range(-1, -len(seq), -1):
            if seq[i] <= 0:
                seq.pop(i)
        return seq
    ```
  - no obvious way to write as **`func = lambda seq: ...`**

Conditional expressions

- Some if-else statements are common and simple enough that they can be condensed into one line, e.g.,

  -
    ```
    def func(x):
        if condition:
            return expr1
        else:
            return expr2
    ```
- What's being returned can be equivalently written in one expression

  -
    ```
    def func(x):
        return expr1 if condition else expr2
    ```
  - `lambda x: expr1 if condition else expr2`
- Terminology
  - see https://docs.python.org/3/reference/index.html sections 6, 7, 8
  - **expression:** a statement that always evaluates to a value/object
  - **statement:** a unit/block of execution
    - all expressions are statements
    - also includes **`var = expr, if/elif/else, for/while, break, continue, def, return, ...`**

Python Comprehensions

Motivation

- Very common pattern to create lists/dicts/sets based on other collections

  -
    ```
    def func(seq):
        result = []
        for item in seq:
            result.append(something about item)
        return result
    ```
- When a repeated pattern shows up, abstract it away!
- Terminology: ***comprehension*** in the sense of ***comprising***
  - e.g., **`result`** above comprises (transformed) objects from the input **`seq`**
  - alternative/related term: **set-builder notation**

List comprehensions

- Syntax
  - `[expr for x in collection]`
  - `[expr for x in collection if test(x)]`
- Equivalent form

  -
    ```
    def make_list(
          collection,
          transform,
          test=lambda x: True,
    ):
          result = []
          for x in collection:
              if test(x):
                  result.append(transform(x))
          return result

    make_list(collection, lambda x: expr, test)
    ```

Other comprehension types

- `dict` comprehension
  - `{`*`key`*`: `*`value`*` for x in `*`collection`*` if `*`test`*`(x)}`
- `set` comprehension
  - `{`*`item`*` for x in `*`collection`*` if `*`test`*`(x)}`
- No `tuple` comprehensions available in Python
  - tuples are immutable, and comprehensions are implicitly "built up"
  - remember: key syntax for tuple literals is the **presence of commas**
  - if you surround a comprehension syntax with parens, Python interprets as a ***generator expression***
    - not covered in 6.100, take 6.101 to learn more
  - to build tuples from other collections, use `tuple(`*`iterable`*`)` syntax, where *`iterable`* can be a comprehension

Next time

- New unit on combinatorial optimization
  - continuation of exhaustive enumeration strategy
  - build on graphs, recursion, DFS