LAMBDAS, DICTIONARIES, AND ABSTRACT DATA TYPES 4

COMPUTER SCIENCE 88

February 17, 2021

1 Lambdas

Lambda expressions are one-line functions that specify two things: the parameters and the return expression.

A lambda expression that takes in no arguments and returns 8:

$$\texttt{lambda:}\quad \underbrace{\texttt{8}}_{\text{return value}}$$

A lambda expression that takes two arguments and returns their product:

$$\texttt{lambda}\quad \underbrace{\texttt{x, y}}_{\text{parameters}}\ \texttt{:}\quad \underbrace{\texttt{x * y}}_{\text{return expression}}$$

Unlike functions created by a `def` statement, the function object that a lambda expression creates has no intrinsic name and is not bound to any variable. In fact, nothing changes in the current environment when we evaluate a lambda expression unless we do something with this expression, such as assign it to a variable or pass it as an argument to a higher order function.

1. What would Python print?

```
>>> a = lambda: 5
>>> a()

>>> a(5)

>>> b = lambda: lambda x: 3
>>> b()(15)

>>> c = lambda x, y: x + y
>>> c(4, 5)

>>> d = lambda x: lambda y: x * y
>>> d(3)

>>> d(3)(3)

>>> e = d(2)
>>> e(5)

>>> f = lambda: print(1)

>>> g = f()
```

2 Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples.[1] Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax `dictionary[key]`.

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:
  ```
  dictionary[key] = val
  ```
- To iterate over a dictionary's keys:
  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```
- To iterate over a dictionary's values:
  ```
  for value in dictionary.values():
      do_stuff()
  ```
- To iterate over a dictionary's keys and values:
  ```
  for key, value in dictionary.items():
      do_stuff()
  ```
- To remove an entry in a dictionary:
  ```
  del dictionary[key]
  ```
- To get the value corresponding to `key` and remove the entry:

[1] To be exact, keys must be *hashable*, which is out of scope for this course. This means that some mutable objects, such as classes, can be used as dictionary keys.

```
dictionary.pop(key)
```

2.1 Questions

1. What would Python display?

```
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'ditto': 25,
    'mew': 151}
>>> 'mewtwo'in pokemon



>>> len(pokemon)



>>> pokemon['ditto'] = pokemon['jolteon']
>>> pokemon[('diglett', 'diglett', 'diglett')] = 51
>>> pokemon[25] = 'pikachu'
>>> pokemon




>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon



>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

2. Given a (non-nested) dictionary `d`, write a function which deletes all occurrences of `x` as a value. You cannot delete items in a dictionary as you are iterating through it.

```
def remove_all(d, x):
    """
    >>> d = {1:2, 2:3, 3:2, 4:3}
    >>> remove_all(d, 2)
    >>> d
    {2: 3, 4: 3}
    """
```

3 Abstract Data Types

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects — for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about `how` code is implemented — they just have to know `what` it does.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

An *abstract data type* consists of two types of functions:

- Constructors: functions that build the abstract data type.
- Selectors: functions that retrieve information from the data type.

For example, say we have an abstract data type called `city`. This `city` object will hold the `city`'s name, and its latitude and longitude. To create a `city` object, you'd use a constructor like

```
city = make_city(name, lat, lon)
```

To extract the information of a `city` object, you would use the selectors like

```
get_name(city)
get_lat(city)
get_lon(city)
```

For example, here is how we would use the `make_city` constructor to create a city object to represent Berkeley and the selectors to access its information.

```
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley'
>>> get_lat(berkeley)
122
>>> get_lon(berkeley)
37
```

The following code will compute the distance between two city objects:

```
from math import sqrt
def distance(city_1, city_2):

    lat_1, lon_1 = get_lat(city_1), get_lon(city_1)
    lat_2, lon_2 = get_lat(city_2), get_lon(city_2)

    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Notice that we don't need to know how these functions were implemented. We are assuming that someone else has defined them for us.

It's okay if the end user doesn't know how functions were implemented. However, the functions still have to be defined by someone. We'll look into defining the constructors and selectors later in this discussion. Notice how we did not need to know how the constructors and selectors in the previous section were implemented in order to use them. This is what we mean by the *implementation* and *use* of an abstract data type being separate. In fact, you should never assume anything about how the constructors and selectors for an abstract data type are implemented. Doing so is called a **data abstraction violation**.

As an example, here is one implemenation for the `rational` constructor.

```
def rational(n, d):
    return [n, d]
```

Given this constructor, the following would be considered a data abstraction violation:

```
>>> frac1 = rational(3, 4)
>>> frac2 = rational(5, 6)
>>> frac1[0] * frac2[0]
15
```

This is because we assumed rationals were represented as lists instead of accessing their elements using the selectors.

3.1 Questions

1. The CS 88 TAs have decided to call upon the power of data abstraction to organize their discussion sections. To do so, they've created a `discussion` abstract data type. A discussion contains three things:

   - The name of the TA running the section
   - The time the section starts, given as an integer
   - A list of students enrolled in the section

   Given this, the TAs come up with the following constructor and selectors:

   - `make_discussion(ta, time, students)`: Creates and returns a new discussion section.
   - `get_ta(disc)`: Returns the TA running the given discussion section.
   - `get_time(disc)`: Returns the start time of the given discussion section.
   - `get_students(disc)`: Returns the list of students enrolled in the given discussion section.

   The TAs have decided to reveal the implementation of the discussion section ADT. Use these function definitions to answer the next two questions:

```
def make_discussion(ta, time, students):
    return [name, time, students]

def get_ta(disc):
    return disc[0]

def get_time(disc):
    return disc[1]

def get_students(disc):
    return disc[2]
```

2. Implement `add_student`, which takes in a discussion section and a string representing a student's name, and returns a new discussion with the new student added to the roster. The list of students for the new discussion should be a new list. Remember to use the constructor and selectors!

```
def add_student(disc, student):
    """ Adds a student to this discussion.
    >>> disc = make_discussion("Alex", 4, ["Srinath", "Brian
       "])
    >>> new_disc = add_student(disc, "Sophia")
    >>> get_students(new_disc)
    ["Srinath", "Brian", "Sophia"]
    >>> get_students(disc)
    ["Srinath", "Brian"]
    """
```

3. A disgruntled student makes changes to the discussion data abstraction in an attempt to disrupt the TAs' ability to run section. The new implementation is as follows:

```
def make_discussion(ta, time, students):
    return {"ta" : ta, "time" : time,  "students" : students}

def get_ta(disc):
    return disc["ta"]

def get_time(disc):
    return disc["time"]

def get_students(disc):
    return disc["students"]
```

   Would the code in the previous question, with the corrections you made, still work with these changes? Would the code before removing abstraction violations still work?