LAMBDAS AND DICTIONARIES 4

DATA C88C

September 18, 2023

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:

```
lambda:  8
```
return value

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

```
lambda  x, y :  x * y
```
parameters — 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 Environment Diagrams

1. Draw the environment diagram for evaluating the following code

```
def mystery_a(lst):
    def mystery_b(color, count):
        lst.extend([color] * count)
    return mystery_b

colors = ["purple", "pink", "brown"]
f = mystery_a(colors)
f("red", 3)
f("blue", 1)
```

2. If on line 2 and line 4, we replace `mystery_b` with `mystery_a`, what will change in the environment diagram, if anything?

3. If on line 3, we change `lst.extend([color] * count)` to `lst.append([color] * count)`, what will change, if anything?

4. Draw the environment diagram for evaluating the following code

```
def ross(geller, num):
    return geller(monica(num))

def monica(num):
    if num >= 2:
        return tup[0]
    return tup[num]


f = lambda x: x[-1] == "a"
tup = ("hola", "there")
rachel = ross(f, 5)
```

5. Draw the environment diagram for evaluating the following code

```
def anna(olaf):
    return lambda a, b: olaf or [a] * b

hans = [1]
elsa = anna(hans.append(4))
kristoff = elsa(3, 4)
```

3 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}
>>> pokemon['mew'] = 15
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 15}
```

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()
  ```

[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.

- 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:
```
dictionary.pop(key)
```

3.1 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. Write a function that takes in a sequence `s` and a function `fn` and returns a dictionary.

   The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. Finally, the key for each value should be `fn(e)`.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {0: [0], 1: [-1, 1], 4: [-2, 2], 9: [-3, 3]}
    """
```