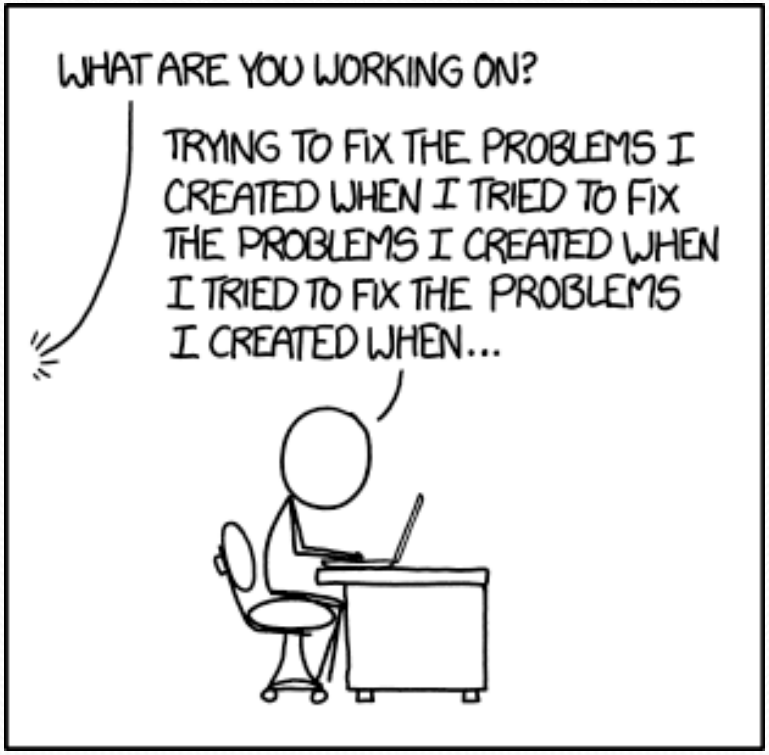
CS 115, Autumn 2021

Lecture 38: lambdas; advanced topics



Thanks to Marty Stepp and Stuart Reges for parts of these slides

Importing

- So far we have used `from` **module_name** `import *` to import modules
  - This can be a problem if we import multiple modules that have functions with the same name
- Another option: `import` **module_name**
  - Imports the module but we must refer to all functions in it as **module_name**.**function_name**(**parameters**)
  - Prevents name conflicts but slightly longer less convenient syntax

Parameters and events

- What happens if we write the following?

```
hello['command'] = print_something("hi")
```

  - This calls the function `print_something` passing it `"hi"` right now – it doesn't set it to be called when `hello` is clicked

  - How can we pass parameters without parentheses?
    - We can't so we need lambda notation

Lambda functions

- Lambda functions – small anonymous function that can be stored in variables
  - You can call another function from within one and have access to the variables in the function you declare it in
  - Example:

```
from ECGUI import *

def print_something(name):
    print("hello " + name.get())
def main():
    window = make_window()
    hello = add_button(window, "hello")
    name = add_entry_box(window)
    hello['command'] = lambda: print_something(name)
    window.mainloop()
```

Advanced Python

- We learned a lot this quarter but there is way more you can do with Python.

- Some topics you may want to explore:
  - tuples
  - dictionaries
  - lists that change size
  - lists of lists
  - list comprehensions
  - classes and objects

Tuples

- A sequence similar to a `list` but it **cannot be altered**
- Good for storing related data
  - We mainly store the same **type** of data in a `list`
  - We usually store related things in tuples

- Creating tuples

name = (**data**, **other_data**, … , **last_data**)

```
tuple = ("Tucson", 80, 90)
```

Using tuples

- You can access elements using [] notation, just like lists and strings

```
tuple = ("Tucson", 80, 90)
low = tuple[1]
```

- You cannot update a tuple!
  - Tuples are immutable
- You can loop through tuples the same as lists

operation	call	result
len()	`len((1, 2, 3))`	`3`
+	`(1, 2, 3) + (4, 5, 6)`	`(1, 2, 3, 4, 5, 6)`
*****	`('Hi!',) * 4`	`('Hi!', 'Hi!', 'Hi!', 'Hi!')`
in	`3 in (1, 2, 3)`	`True`
for	`for x in (1,2,3): print x,`	`1 2 3`
min()	`min((1, 3))`	`1`
max()	`max((1, 3))`	`3`

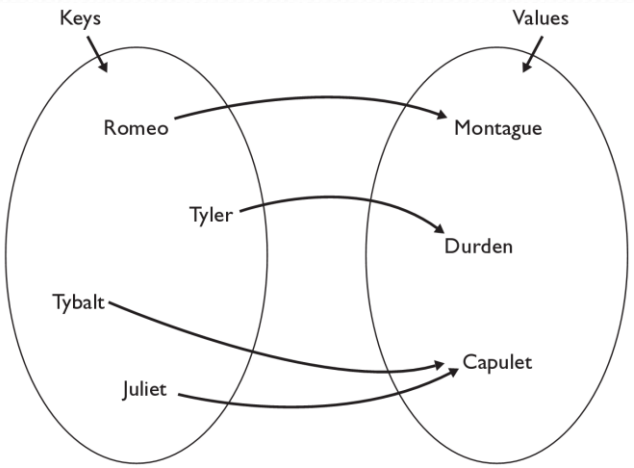
Exercise

- Write a program to count the number of occurrences of each unique word in a large text file (e.g. *Moby Dick*).
  - Allow the user to type a word and report how many times that word appeared in the book.
  - Report all words that appeared in the book at least 500 times.
- What structure is appropriate for this problem?

Dictionaries

- **dictionary**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "map", "associative array", "hash"

- basic dictionary operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



`my_dict["Juliet"]` returns `"Capulet"`

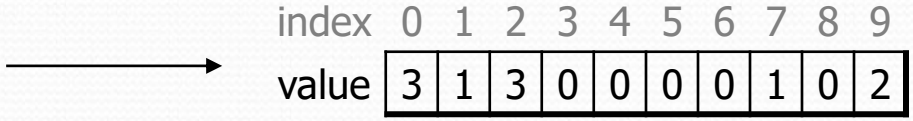
`Dictionary` functions

`my_dict[`**key**`] = `**value**	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`my_dict[`**key**`]`	returns the value mapped to the given key (error if key not found)
`items()`	return a new view of the dictionary's items ((key, value) pairs)
`pop(`**key**`)`	removes any existing mapping for the given key and returns it (error if key not found)
`popitem()`	removes and returns an arbitrary (key, value) pair (error if empty)
`keys()`	returns the dictionary's keys
`values()`	returns the dictionary's values

You can also use `in`, `len()`, etc.

Dictionaries and tallying

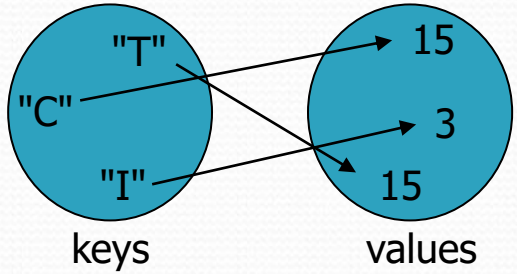
- a `dictionary` can be thought of as generalization of a tallying `list`
  - the "index" (key) doesn't have to be an `int`
  - count digits: `22092310907`




```
# (T)rump, (C)linton, (I)ndependent
```

  - count votes:

```
"TCCCCCCTTTTTCCCCCCTCTTITCTTITCCTIC"
```



items, keys and values

- `items` function returns tuples of each key-value pair
  - can loop over the keys in a for loop

```
ages = {}
ages["Merlin"] = 4
ages("Chester"] = 2
ages["Percival"] = 12
for cat, age in ages.items()):
    print(name + " -> " + str(age))
```

- `values` function returns all values in the dictionary
  - no easy way to get from a value to its associated key(s)
- `keys` function returns all keys in the dictionary

Exercise: Mountain peak

Write a program that reads elevation data from a file, draws it on a `drawing_panel` and finds the path from the highest elevation to the edge of the region and draws that path on the `drawing_panel`.

Data:

```
34 76 87 9 34 8  22 33 33 33 45 65 43 22
5  7  88 0 56 76 76 77 4  45 55 55 4  5
…
```

This data is a different shape. How should we store it?

Lists of lists

- You can put a list in a list

```
list = [[1, 2, 3], [4, 5, 6]]
```

How can you access 2?

```
list[0][1]
```

How can you find the length of the second inner list (`[4, 5, 6]`)?

```
len(list[1])
```

Lists of lists - traversals

- We normally use nested loops to go through a list of lists

```
data = [[1, 2, 3], [4, 5, 6]]
for i in range(len(data)):
    for j in range(len(data[i])):
        # do something with data[i][j]
```

Why does are inner loop go to `len(data[i])`?

List of lists mystery

```
def mystery(data, pos, n):
    result = []
    for i in range(0, n):
        for j in range(0, n):
            result.append(data[i + pos][j + pos])
    return result
```

Suppose that a variable called grid has been declared as follows:

```
grid = [[8, 2, 7, 8, 2, 1], [1, 5, 1, 7, 4, 7],
        [5, 9, 6, 7, 3, 2], [7, 8, 7, 7, 7, 9],
        [4, 2, 6, 9, 2, 3], [2, 2, 8, 1, 1, 3]]
```

which means it will store the following 6-by-6 grid of values:

8	2	7	8	2	1
1	5	1	7	4	7
5	9	6	7	3	2
7	8	7	7	7	9
4	2	6	9	2	3
2	2	8	1	1	3

For each call at right, indicate what value is returned. If the function call results in an error, write error instead.

Function Call	Contents of List Returned
`mystery(grid, 2, 2)`	_____
`mystery(grid, 0, 2)`	_____
`mystery(grid, 3, 3)`	_____

List comprehensions

```
[expression for element in list]
```

- A compact syntax that can replace loops that alter lists
  - Applies the expression to each element in the list
  - You can have 0 or more `for` or `if` statements

List comprehensions

```
vec = [2, 4, 6]
result = [3 * x for x in vec]
print(result)                  # [6, 12, 18]

result2 = [3 * x for x in vec if x > 3]
print(result2)                 # [12, 18]

result3 = [3 * x for x in vec if x < 2]
print(result3)                 # []
```

Notice the contents of `vec` do not change

List comprehensions

More than one element can be generated from each element in the original list

```
vec = [2, 4, 6]
result = [[x, x ** 2] for x in vec]
print(result)                  # [[2, 4], [4, 16], [6,
36]]

result2 = [(x, x ** 2) for x in vec]
print(result2)                 # [(2, 4), (4, 16), (6,
36)]
```