Sets and Dictionaries (Ch. 9)

CSC110: INTRO TO COMPUTER PROGRAMMING WITH PYTHON

INSTRUCTOR: BILL BARRY

1

R
o ☞
a
d
m
a
p

Sets & Dictionaries

All project, quiz, and lab work due Fri 11:59pm

NSC finals Tues-Thurs

9am class: Tues, 8am

10am class: Thurs, 10:30am

2

This Week: Sets & Dictionaries

✓ Intro to Collections (Mon)

✓ Sets (Mon)

✓ Dictionaries (Tues)

✓ Intro to Objects (Wed)

Looking Ahead: Language Differences, CSC142 (Thurs)

Final Prep (Fri)

3

Intro to Collections

4

Python Collections

A **collection** is an object that manages elements; it’s a more general category than *sequences*, which are always *ordered*

Python offers several different collections, including:

- Lists
- Sets
- Dictionaries

Typical behaviors offered by Collections:

- Ability to **add or remove** elements
- Are **iterable** (can be used in a `for` loop to consume elements)
- Have **mutable elements**; when passed to functions, affects caller’s version, too

Overview of Collections: https://www.sitepoint.com/a-tour-of-python-collections

We’ve already looked at Strings and Lists; let’s explore the other two

5

Sets

6

Sets

Analogy

- You are collecting all the Dust Bunny comics from 1980
- A friend brings you Dust Bunny 1980 #3; you buy it
- Another friend brings you another Dust Bunny 1980 #3; you politely refuse
- In your collection you have whatever issues you can accumulate, but no duplicates

Purpose

- Manage an unordered set of unique values

Examples

- You might want to get a list of words Shakespeare used in Hamlet
- …but you don't care about duplicates, just a list of distinct words
- Process: feed all words to a set; only unique words are kept

7

Working with Sets #1

Create

- Use the { } delimiters, e.g., `greetings = {'hello', 'bye', 'hey'}`
- …or pass in an iterable, e.g., `greetings = set(['hello', 'bye', 'hey'])`
- For empty, call the set function, e.g., `greetings = set( )`
- For other examples and oddities, see p. ~463

Count

- Use len function, e.g.: `len(greetings)`

Add

- Use add method, e.g.: `greetings.add('howdy')`
- …but duplicates won't be added

Remove

- Use the remove method, e.g.: `words.remove('hey')`

Be aware that { } can create a set or a dictionary, so set( ) is a guarantee you get what you intend

8

Working with Sets #2

Iterate

- for loops will iterate over a set
- Example: **`for`** `greeting` **`in`** `greetings:`
- …but ordering is indeterminate; it's not an ordered collection

Membership

- Use `in` and `not in` to check membership, e.g.: `if 'howdy'` **`in`** `words:`

Conversion

- Use `list()` to convert a set into a list, e.g., **`list`**`(greetings)`

Lots more is offered…

- …including unions, intersections, differences, subsets, supersets, etc.
- …but you don't need to know these additional topics for this course

9

Pause & Play

LET'S DO A CHALLENGE ON SETS

10

Set Exercise: Goodnight Moon

Use supplied text file: GoodnightMoon.txt

- This contains the text of the popular children's book Goodnight Moon by Margaret Wise Brown
- Each word is on its own line
- There is no punctuation or capitalization

Use a **set** to find out how many distinct words are in the book; print that

Also print out the set of unique words in 4 columns

Challenge: write a function called printSetInColumns that takes two parameters, the set to print and the number of columns desired. Assume display width of 80 characters.

Do you expect the printed list of words to be sorted? Do you expect them to be in the same order they appeared in the file?

11

Goodnight Moon: Code

```
# create empty set, add words from file to it
distinctWords = set()
goodnightFile = open("GoodnightMoon.txt", "r")
for oneLine in goodnightFile:
    distinctWords.add(oneLine.strip())
goodnightFile.close()

# print report
print("Number of unique words: ", len(distinctWords), "\n")
wordOnLine = 0
for word in distinctWords:
    print(format(word, "20s"), end = "")
    wordOnLine += 1
    if wordOnLine % 4 == 0:
        print()
```

This is a great example of a scenario for which we'd use Python to solve a problem

12

Goodnight Moon: Challenge

```
DISPLAY_WIDTH = 80

def printSetInColumns(theSet, numColumns):
    columnWidth = DISPLAY_WIDTH // numColumns
    itemFormat = str(columnWidth) + "s"
    itemOnLine = 0
    for setItem in theSet:
        print(format(str(setItem), itemFormat), end = "")
        itemOnLine += 1
        if itemOnLine % numColumns == 0:
            print()

# ...set creation/file reading code here...

# print report
print("Number of unique words: ", len(distinctWords), "\n")
printSetInColumns(distinctWords, 4)
wordOnLine = 0
```

Does this only work with sets?

Why the str() call here?

13

Dictionaries

14

Dictionary

Analogy

- To use an English dictionary, look up a *word* to find its *definition*
- You could say the dictionary *maps* words to definitions

Purpose

- A Python dictionary lets you store *key/value pairs*
- These keys and values can be of any data type
- You add key/value pairs to the dictionary and can later supply the *key* to find the corresponding *value*
- Depending on language implementation, this can be done extremely quickly

Examples

- Store friend names and birthdays
- Store student's IDs and their GPAs
- Store customer numbers and balances

Note: these are called "maps" in some languages like Java

15

Working with Dictionaries #1

Create

- Use **curly braces** around the elements
- Put a **colon** between the *key* and the *value*
- Put a **comma** between key/value pairs
- Example: `friendAges = {'Chris' : 29, 'Pat' : 47, 'Avery' : 36}`

Retrieve ("look up")

- Use the name of the dictionary variable, then the **key in square brackets**
- Example: `friendAges['Chris']          # returns 29`

Existence

- Find out if a **key** is in/not in the dictionary using **`in`** –and– **`not in`**
- Example: `if 'Pat' in friendAges:`

16

Working with Dictionaries #2

Add

- Use **square bracket** notation with the new **key** inside; assign this to the new **value**
- Example: `friendAges['Lupe']  = 42`

Delete

- Use the **del** keyword
- Example: `del friendAges['Lupe']`

Count

- Use **len** function
- Example: `len(friendAges)    # returns 3`

Working with Dictionaries #3

Mixing types

- Any data types will work in a dictionary
- The key could be a number, string, etc.
- The value could be any of those, but also a list
- Example: `empWorkDays = { 'Parker' : ['M', 'W', 'Th', 'S'], ... }`

Map employees to work schedules

Iterate

- A for loop can easily loop through a dictionary's **keys** (not values)
- Example: `for keyVar in friendAges: # key will be 'Chris', 'Pat', then 'Avery'`

Lots more methods are offered

- …but you don't need to know them for now

Pause & Play

LET’S DO A CHALLENGE ON DICTIONARIES

19

Dictionary Exercise: Expenses

Create a dictionary that maps an expense category to a list of expenses for that category

Here’s the data to use:

- Rent → 1500
- Utilities → 53.75, 113.82, 206.96
- Food → 104.56, 22.18, 85.49, 8.99
- Eating Out → 67.23, 25.17

```
Rent            $  1,500.00
Utilities       $    374.53
Food            $    221.22
Eating Out      $     92.40
---------------   ---------
TOTAL           $  2,188.15
```

Then, write code that displays each category and the total of the expenses in that category (nicely formatted, of course). At the bottom, provide a grand total for all expenses

No functions are necessary; just write the code

20

Expenses: Code

```
expenseDict = { "Rent"        : [1500],                              \
                "Utilities"   : [53.75, 113.82, 206.96],             \
                "Food"        : [104.56, 22.18, 85.49, 8.99],        \
                "Eating Out" : [67.23, 25.17]                        \
              }

grandTotal = 0.0
for category in expenseDict:
    categoryTotal = 0.0
    amountList = expenseDict[category]
    categoryTotal = sum(amountList)
    print(format(category, "15s"), "$", format(categoryTotal, "9,.2f"))
    grandTotal += categoryTotal

print(           "-" * 15,    " ",               "-" * 9)
print(format("TOTAL", "15s"), "$", format(grandTotal, "9,.2f"))
```

```
Rent            $  1,500.00
Utilities       $    374.53
Food            $    221.22
Eating Out      $     92.40
---------------   ---------
TOTAL           $  2,188.15
```

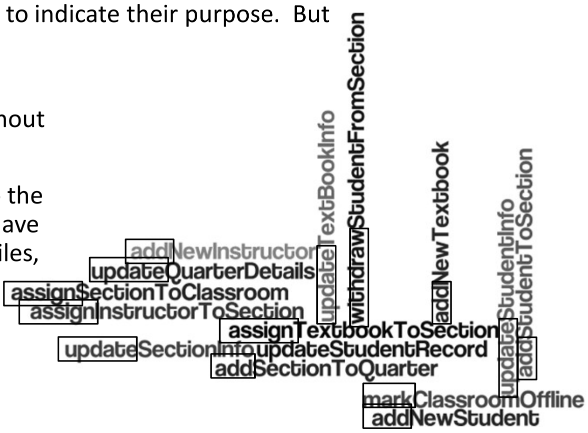
Intro to Classes and Objects

Procedural vs. Object Programming

In Procedural programming (what we’ve done so far), we focus on writing **functions that do things**. We even tend to name them with **verbs** to indicate their purpose. But verbs don’t help much with organizing our code

Consider the data-processing system at North. We might expect to find many functions throughout a complex set of interrelated programs

The issue? If you make fundamental changes to the way student data is handled, where might you have to look? The code might be strewn over many files, over many systems, and be nearly impossible to locate. It would be prone to bugs and hard to maintain!



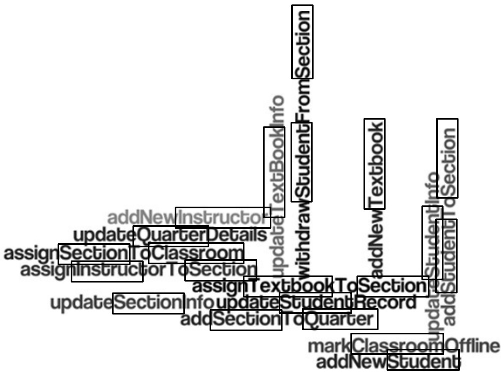
23

Reorganizing Our Thoughts

But what if everything related to a particular “actor” or “entity” in the system were together in one place? Just look at the word cloud diagram again and focus your eyes on the **nouns** rather than the verbs; see what emerges. Here are the **objects** (“entities”) we might identify:

Student
Section
Textbook
Class
Instructor
Classroom
Quarter

Now, if you need to change something about a student, it’s clear where to look. Need to add code? You’ll have a clear idea where to start



24

Classes and Objects

A **class** is analogous to a **recipe for a cake**

- It's a nice plan; you can read it, see what it will require, etc.
- But you can't frost it, cut it, or know exactly how it will taste; it's just a concept

An **object** is analogous to a **cake made from the recipe**

- It's real; you can smell it, taste it, etc.
- You can also test it; did the ingredients come together in a harmonious way? Did it rise properly? Does the recipe need some adjustments?

In code, we write classes, then *instantiate* objects from them; this breathes life into the object and makes it a real, in-memory entity we can interact with

25

Behaviors and State

Objects have **behaviors** (methods) associated with them

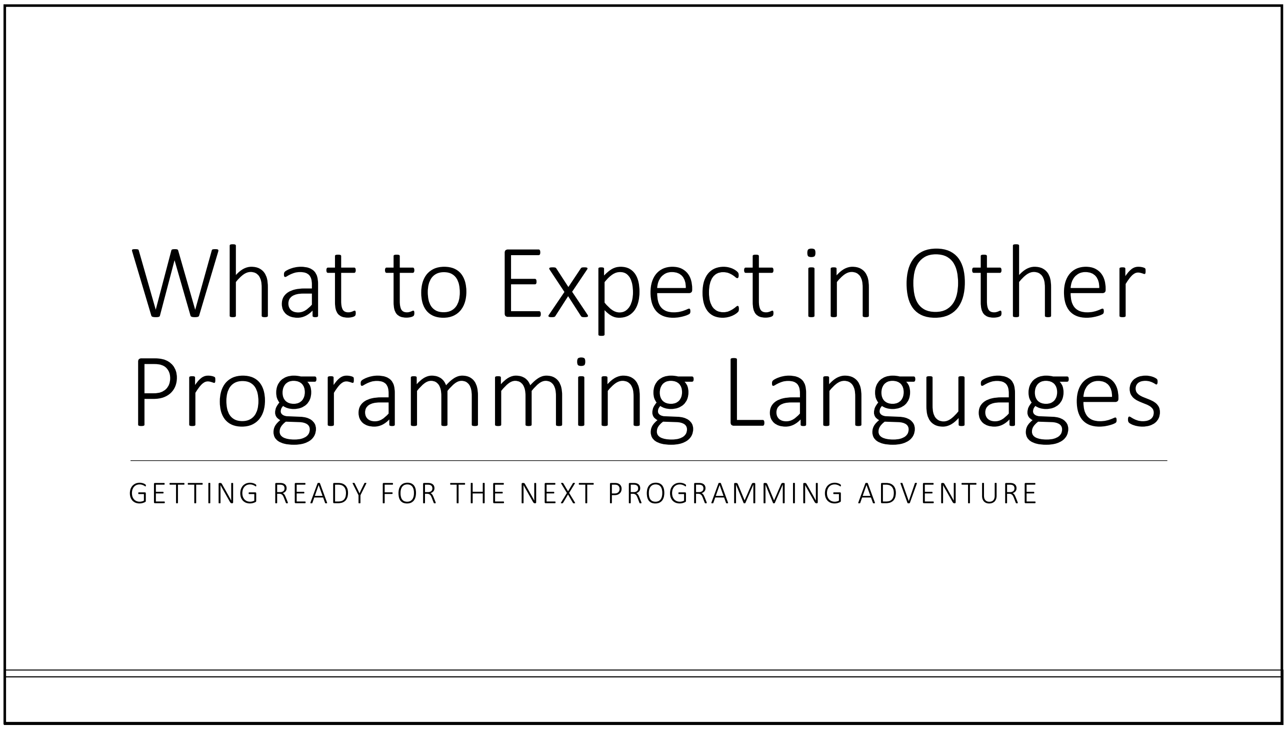
- Example: cakes can be frosted, decorated, and cut into slices
- Implementation: behaviors are implemented via **methods**

Objects have **state** (data) associated with them

- Example: the cake is either whole or has been cut; it can be boxed (for delivery) or unboxed; it can have a certain number of candles on it (for celebrations)
- Implementation: state is implemented via **class-level variables**

Think of what kinds of "conversations" you want to have with the cake; what do you want to ask it to do, what does it "remember?"

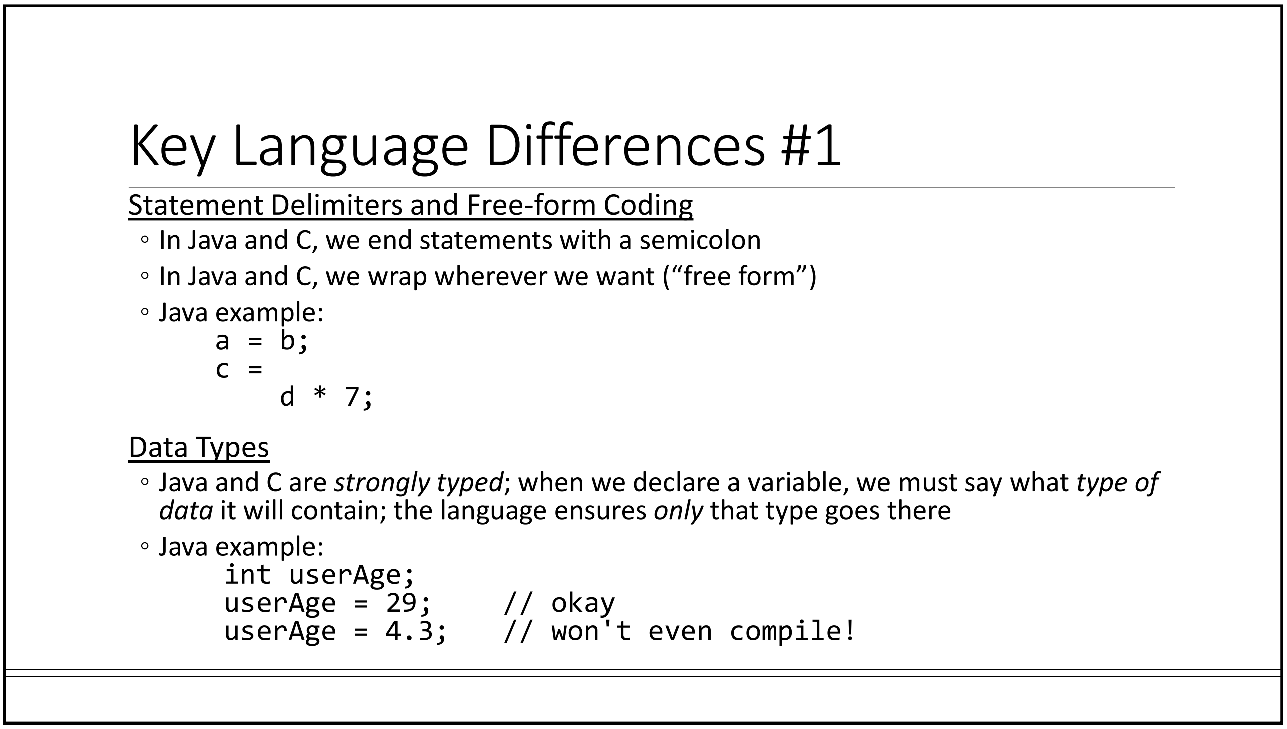
26



What to Expect in Other Programming Languages

GETTING READY FOR THE NEXT PROGRAMMING ADVENTURE

27



Key Language Differences #1

Statement Delimiters and Free-form Coding

- In Java and C, we end statements with a semicolon
- In Java and C, we wrap wherever we want (“free form”)
- Java example:

```
a = b;
c =
    d * 7;
```

Data Types

- Java and C are *strongly typed*; when we declare a variable, we must say what *type of data* it will contain; the language ensures *only* that type goes there
- Java example:

```
int userAge;
userAge = 29;    // okay
userAge = 4.3;   // won't even compile!
```

28

Key Language Differences #2

Conditions Go in Parentheses

- In many languages (e.g., Java, C, JavaScript), conditions must go in parentheses
- Example: `if (age > 21)`

Code Blocks

- To identify code blocks, Java and C use curly braces instead of Python's colon and indentation
- Java example:

```
if (age < 16) {
    System.out.println("No driver's license allowed")
} else {
    System.out.println("License testing permitted")
}
```

29

Key Language Differences #3

Objects

In Java, OOP is fundamental and pervasive

- All code we write is within a class
- We're constantly creating (instantiating) and using objects
- All *functions* are therefore *methods*

Parameter Passing

- Feels like Python in most ways, though the foundations are different
- When passing basic data ("primitives"), functions can't affect the caller's data
- When we pass objects to a function (e.g., lists), object data is mutable; the function can affect the caller's data

30

What to Expect in CSC142

IF YOU'RE MOVING FORWARD…

31

What to Expect in CSC142

Coverage of key Java syntax. Quick review of what you already know, applied to Java (expressions, control structures, functions, etc.)

Emphasis on OOP, introduced then integrated with other topics

Significant projects with less "scaffolding" provided; these will require smart planning and coding skills, or you'll spin your wheels and/or end up in the ditch

Utilization and refinement of debugging skills

Introduction to design documentation, drawings that can help you create and navigate an OOD/OOP world

Labs from UW's Practice-It site: https://practiceit.cs.washington.edu/ with screenshots of completed lab submitted for grading

32

Projects

1. Java basics and definite loops
2. Graphics and using Java-provided objects
3. Indefinite loops, user input, data type and range validation, preconditions
4. Creating and using classes and objects
5. Arrays of objects
6. Inheritance, polymorphism, abstracts
7. ArrayList and file I/O

33

The End

34