Programming I

Course 7

Introduction to programming

What did we talk about last time?

- Functions
- Interfaces
- Code Abstraction

What will we speak about?

TESTING

ASSERTIONS

DEBUGGING

EXCEPTIONS

QUALITY?

- You are cooking soup and find bugs inside the pot.
- What can you do?

QUALITY?

- You are cooking soup and find bugs inside the pot.
  - check soup for bugs again
    - Reproduce the issue
    - Find source of bugs

TESTING

QUALITY?

DEFENSIVE PROGRAMMING

- You observe bugs falling from the ceiling.
- Keep lid on pot while cooking
  - You know about the bugs and you guard against them

QUALITY?

DEBUGGING

- You observe bugs falling from the ceiling.
- Call exterminator
- Eliminate the source of bugs

TESTING
or
VALIDATION

- Compare input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”

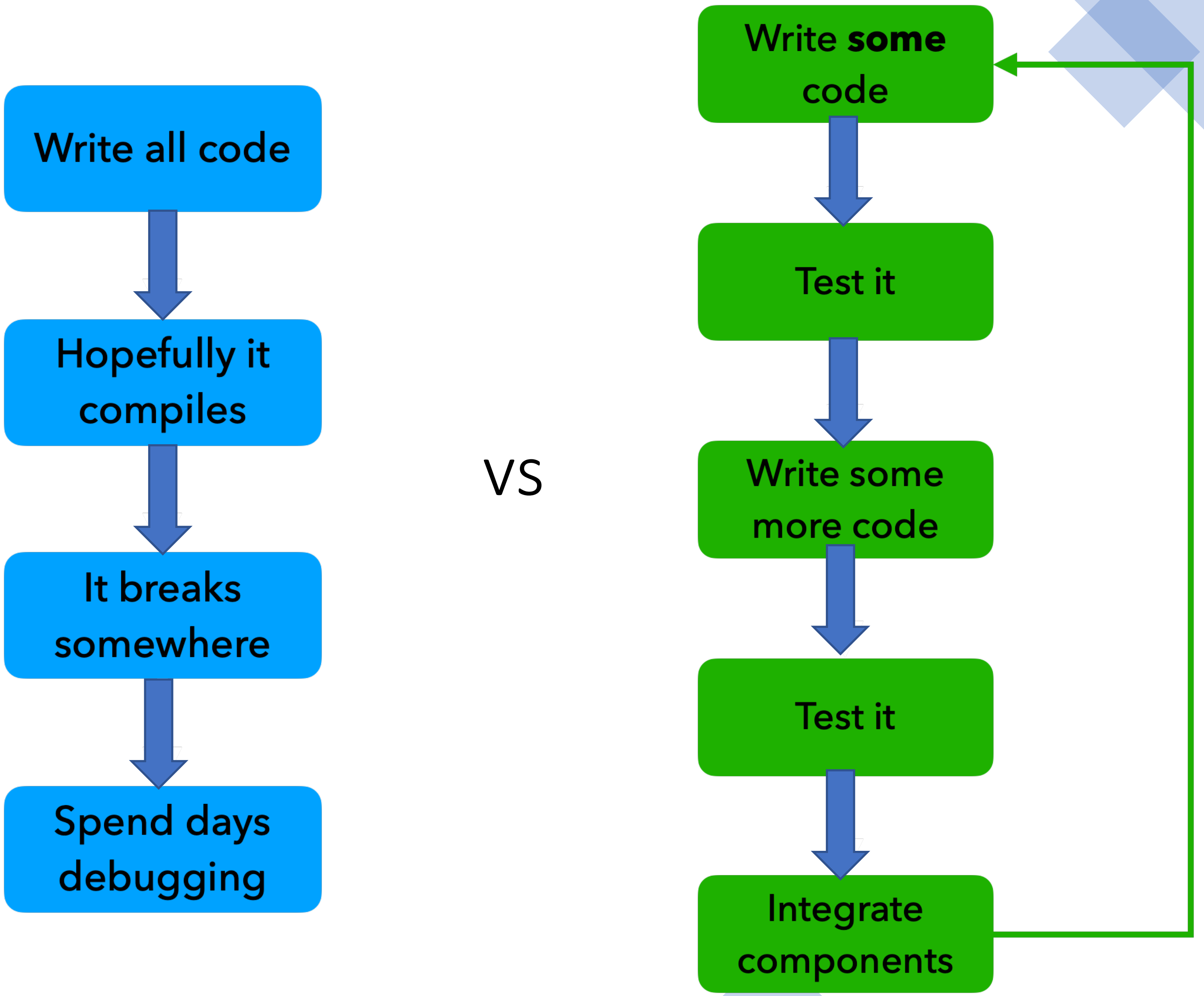
DEFENSIVE PROGRAMMING

- Write specifications for functions
- Write modular programs
- Check conditions on inputs/outputs (assertions)

DEBUGGING

- Study events leading up to an error
- “Why is it not working?”
- “How can I fix my program?”

Prepare Code for Testing and Debugging



Prepare Code for Testing and Debugging

- Break program up into functions and modules that can be tested and debugged individually
- Document constraints on modules
  - What do you expect the input to be?
  - What do you expect the output to be?
- Document assumptions behind code design

Are you ready to test?

- Ensure code runs
  - Remove syntax errors
  - Remove static semantic errors
  - Python interpreter or IDE can usually find these for you

Are you ready to test?

- Have a set of expected results
  - An input set
  - For each input, the expected output
- Think of some situations that could break your code

Who writes the tests?

The developer

- Some tests that he/she thinks are relevant for common usage of the program

The test engineer

- Edge cases
- Unexpected inputs
- Tries to make the developer cry

Why have a test engineer?

Let's look at a problem from user point of view

Requirments

- Adding two numbers of max two digits

Expected behaviour

- The program will read two numbers and will print the sum.
- The user has to press ENTER after each number.

Step1 – Simple test

Purpose

- familiarizing with the program

How?

- Check minimal program stability: program often crashes right away
- Do not spend too much time
- Start the program and add 2 with 3

Result of Step 1

Program

?2

?3

5

? ..

Problems?

- Nothing shows what program this is
- No onscreen instructions
- How to stop the program?
- Numbers alignment

Actions

- Create problem reports
- One problem per report

Report

Report

- Report type (coding, design, suggestion, documentation, hardware, query)
- Severity (fatal/serious/minor)
- Problem summary
- Is reproducible? Steps to reproduce
- Problem description
- Suggested fix (optional)
- Reported by
- Date

Report

Report #1

- Report type: design
- Severity serious
- Problem summary: User can not understand what the program requests as input
- Reproducibility: start the program
- Problem description: Program should tell user what input is currently asking for
- Suggested fix: add message when asking for input
- Reported by name@it.com
- Date 10/NOV/2021

Step 2 – Some other functional tests

- Valid inputs using all digits:
  - 99+99
  - -99+ -99
  - 99+-14
  - -38+99
  - 56+99
  - 9+9
  - 0+0
  - 0+23
  - -78+0
  - Etc.

Step 3 – Tests planning

- Identify classes of tests:
  - if the same result is expected from two tests, test only one of them
- Tests the variant most likely to fail
  - look at the boundaries of a class
- Finding boundary conditions
  - no magic way, use experience
- Test both
  - Programming boundaries (from program view)
  - testing boundaries (user perspective)

Step 4 – Explore Invalid cases

- Switching from formal to informal tests
  - when program crashes unexpectedly (it's ok to use print)
- Keep testing with invalid cases
- No formality needed as the program may have to be redesigned
  - always write down the results

Step 5 – Summary of Behaviour

For tester's use

- Helps thinking about the program in order to elaborate a testing strategy later
- Identify new things like edge cases

Example

- The program does not deal with negative numbers
- The program accepts any char as a valid input until <Enter>
- The program does not check if some number is entered before <Enter>

Failure causes

- Partial failure is inevitable
  - Goal: prevent complete failure
  - Structure your code to be reliable and understandable

Failure causes

- Some failure causes:
  - Misuse of your code
    - Precondition violation
  - Errors in your code
    - Bugs, representation exposure, many more
  - Unpredictable external problems
    - Out of memory
    - Missing file
    - Memory corruption

Classes of Tests

- Unit testing
  - validate each piece of program
  - testing each function separately
- Regression testing
  - add test for bugs as you find them
  - catch reintroduced errors that were previously fixed
- Integration testing
  - does overall program work?

Testing Approaches

- Intuition about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else
    False """
```

  - can you come up with some natural partitions?

- If no natural partitions, might do random testing
  - probability that code is correct increases with more tests

Testing Approaches

- Black box testing
  - explore paths through specification
  - User
- Glass/white box testing
  - explore paths through code
  - programmer

Black Box Testing

```
def sqrt(x, eps):
        """ Assumes x, eps floats, x >= 0, eps > 0

        Returns res such that x-eps <= res*res <= x+eps
        """
```

- Designed without looking at the code
  - can be done by someone other than the implementer to avoid some implementer biases
- Testing can be reused if implementation changes
- Paths through specification
  - build test cases in different natural space partitions
  - also consider edge cases (empty lists, singleton list, large numbers, small numbers)

Black Box Testing

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	X	EPS
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.25	0.0001
irratinal sqare root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

White Box Testing

- Use code directly to guide design of test cases
- Called path-complete if every potential path through code is tested at least once
- What are some drawbacks of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths

White Box Testing

- Guidelines
  - branches
  - for loops
  - while loops

Test all branches of a conditional statement

Test:
- Loop body not entered
- Loop body executed once
- Loop body executed multiple times

White Box Testing

```
def abs(x):
        """ Assumes x is an int
        Returns x if x>=0 and –x otherwise """
        if x < -1:
                return –x
        else:
                return x
```

- a path-complete test suite could miss a bug
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test edge cases

Debugging

- steep learning curve
- goal is to have a bug-free program
- Tools
  - built in to IDE and Anaconda
  - Python Tutor
  - print statement (loggers)
  - be systematic in your hunt

Print Statements

- Good way to test hypothesis
- When to print
  - Enter function
  - Parameters
  - Function results
- Use bisection method
  - put print halfway in code
  - decide where bug may be depending on values

Debugging Steps

- Study program code
  - don’t ask what is wrong
  - ask how did I get the unexpected result
  - is it part of a family?

- Scientific method
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to test with

Error Messages - Easy

- Trying to access beyond the limits of a list

```
test = [1,2,3]
test[4]
```
→ `IndexError`

- Trying to convert an inappropriate type

```
int(test)
```
→ `TypeError`

- Referencing a non-existent variable

```
a
```
→ `NameError`

- Mixing data types without appropriate coercion

```
'3'/4
```
→ `TypeError`

- Forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)
```
→ `SyntaxError`

Logic Errors - Hard

- Think before writing new code
- Draw pictures, take a break
- Explain the code to
  - someone else
  - a rubber duck

Assertions

- Assumptions on the state of computation
- Use an assert statement to raise an `AssertionError` exception if assumptions not met
- An example of good defensive programming

Assertions

```
assert condition[, message]
```

```
assert sqrt(4)==2
```

```
assert sqrt(9)==3, ’Sqrt(9) should be 3’
```

Assertions

- Check
  - Precondition
  - Postcondition
  - representation invariant
  - other properties that you know to be true
- Check statically via reasoning (& tools)
- Check dynamically at run time via assertions

```
assert index >= 0;
assert size % 2 == 0, “Bad size for list”
```

- Write the assertions as you write the code

Do not use Assertions for Defensive Programming

- assertions don’t allow a programmer to control response to unexpected conditions
- ensures that execution halts whenever an expected condition is not met
- can be used to check inputs to functions, but not recommended
  - We will talk about Exceptions
- can be used to check outputs of a function (testing)
- should make it easier to locate a bug (automated)

Exceptions

- What happens when procedure execution hits an unexpected condition?
- Get an exception... to what was expected
  - Trying to access beyond the limits of a list

    `test = [1,2,3]`
    then `test[4]` → `IndexError`

  - Trying to convert an inappropriate type
    `int(test)` → `TypeError`

  - Referencing a non-existent variable
    `a` → `NameError`
  - Mixing data types without appropriate coercion
    `'3'/4` → `TypeError`
  - Forgetting to close parenthesis, quotation, etc.

    ```
    a = len([1,2,3]
    print(a)
    ```
    → `SyntaxError`

When code reaches an unexpected state...

- Trying to access beyond the limits of a list

```
test = [1,2,3]
test[4]                                    → IndexError
```

- Trying to convert an inappropriate type

```
int("test")                                → TypeError
```

- Referencing a non-existent variable

```
print(a)                                   → NameError
```

- Mixing data types without appropriate coercion

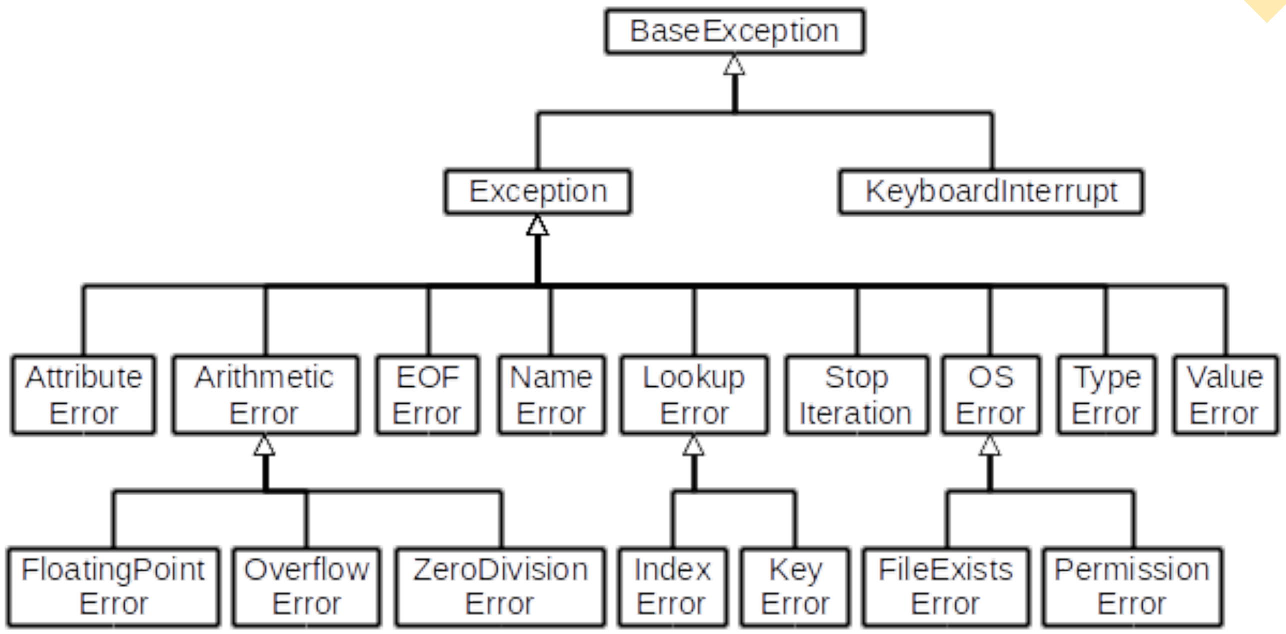
```
'3'/4                                      → TypeError
```

- Forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)                                   → SyntaxError
```

Other Types of Errors

- `SyntaxError`: Python can't parse program
- `NameError`: local or global name not found
- `AttributeError`: attribute reference fails
- `TypeError`: operand doesn't have correct type
- `ValueError`: operand type okay, but value is illegal
- `IOError`: IO system reports malfunction (e.g. file not found)



Dealing with Exceptions

- Python code can provide handlers for exceptions

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

- Exceptions raised by any statement in body of try are handled by the except statement and execution continues with the body of the except statement

Handling Specific Exceptions

- Have separate except clauses to deal with a particular type of exception

```
try:
        a = int(input("Tell me one number: "))
        b = int(input("Tell me another number: "))
        print("a/b = ", a/b)
        print("a+b = ", a+b)
except ValueError:
        print("Could not convert to a number.")
except ZeroDivisionError:
        print("Can't divide by zero")
except:
        print("Something went very wrong.")
```

Only execute if this errors come up

For all others errors

Other try clauses

- `else`:
  - body of this is executed when execution of associated *try* body completes with no exceptions
- `finally`:
  - body of this is always executed after *try*, *else* and *except* clauses, even if they raised another error or executed a *break*, *continue* or *return*
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

What to do with exceptions?

- Fail silently
  - use default values or just continue • bad idea! user gets no warning
- Return an “error” value
  - what value to choose?
  - complicates code having to check for a special value
- Stop execution, signal error condition
  - in Python: raise an exception

```
raise Exception("descriptive string")
```

Exceptions as Control Flow

- Don’t return special values when an error occurred and then check whether ‘error value’ was returned
  - instead, raise an exception when unable to produce a result consistent with function’s specification

```
raise <exceptionName>(<arguments>)

raise ValueError("something is wrong")
```

Keyword

Name of the error you want to raise

Optional by typically a string with a message

Example

```
def get_ratios(L1, L2):
        """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
        ratios = []
        for index in range(len(L1)):
                try:
                        ratios.append(L1[index]/L2[index])
                except ZeroDivisionError:
                        ratios.append(float('nan')) #nan = not a number
                except:
                        raise ValueError('get_rations called with bad arg')
        return ratios
```

Manage flow of program by raising own error

Example of exceptions

- assume a class list for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a new class list, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]]
```

Example

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):
      new_stats = []
      for elt in class_list:
            new_stats.append([elt[0], elt[1], avg(elt[1])])
      return new_stats


def avg(grades):
      return sum(grades)/len(grades)
```

Error if no Grade for a Student

- if one or more students don’t have any grades, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
[['bruce', 'wayne'], [10.0, 8.0, 74.0]],
[['captain', 'america'], [8.0,10.0,96.0]], [['deadpool'], []]]
```

- get `ZeroDivisionError`: float division by zero because try to

```
return sum(grades)/len(grades)
```

Length is 0

Solution: Flag the Error by Printing a message

- decide to notify that something went wrong with a msg

```
def avg(grades):
        try:
                return sum(grades)/len(grades)
        except ZeroDivisionError:
                print('warning: no grades data')
```

Flagged the error

- running on some test data gives

```
worning: no gardes data
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

Because `avg` did not return anything in the except

Solution: Change the Policy

- decide to notify that something went wrong with a msg

```
def avg(grades):
        try:
                return sum(grades)/len(grades)
        except ZeroDivisionError:
                print('warning: no grades data')
                return 0.0
```

Still flag the error

- running on some test data gives

```
worning: no gardes data
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

Now `avg` returns 0

Using assertions as argument checks

```
def avg(grades):
      assert len(grades) != 0, 'no grades data'
      return sum(grades)/len(grades)
```

- raises an `AssertionError` if it is given an empty list for grades
- otherwise runs ok

Exceptions in Review

- Use an exception when
  - Used in a broad or unpredictable context
  - Checking the condition is not feasible
  - Example (transforming a string into a integer, checking is cumbersome, trying is easy)
- Use a precondition (documentation contract) when
  - Checking would be prohibitive (requiring that a list be sorted)
  - Used in a narrow context in which calls can be checked
  - Example (check if list has duplicates, checking is cumbersome, trying does not help, probably not returning the correct result)

Exceptions in Review

- Avoid preconditions because
  - Caller may violate precondition
  - Program can fail in an uninformative or dangerous way
  - Want program to fail as early as possible
- How do preconditions and exceptions differ, for the client?

Exceptions in Review

- Use checked exceptions most of the time
- Handle exceptions sooner rather than later
- Not all exceptions are errors
  - A program structuring mechanism with non-local jumps
  - Used for exceptional (unpredictable) circumstances

Project

- https://docs.google.com/document/d/1YbTR8d9C9FMtR8jRLIpESkmPEfGjZLvlllk29Vacbnw/edit?usp=sharing

Bibliography

- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/