Incremental Development, Test, Evaluation

Problem Solving using Python - Week 4

Week 4 - Learning Objectives

Week 4 - Learning Objectives

You will be able to

Week 4 - Learning Objectives

You will be able to

1. ... solve programming problems using the *incremental development* methodology.

Week 4 - Learning Objectives

You will be able to

1. ... solve programming problems using the *incremental development* methodology.
2. ... preform well the *test* phase.

Week 4 - Learning Objectives

You will be able to

1. ... solve programming problems using the *incremental development* methodology.
2. ... preform well the *test* phase.
3. ... evaluate a program code by the *functionality* criterion.

Incremental Development Methodology

Programming Problem Solving Model

Programming Problem Solving Model

1. Reinterpret the Problem
2. Design a Solution
3. Code
4. Test
5. Debug
6. Evaluate & Reflect

Our goal is to develop a function that calculate the distance between two points `(x1, y1)` and `(x2, y2)`, by the Pythagorean theorem:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

This problem is deliberately simple so that we can focus on important **methodology** in problem solving.

Incremental Development

- The goal is to avoid long debugging sessions by adding and testing only a small amount of code at a time
- We will still use that **Programming Problem Solving model**
- But we will do a few cycles of **Problem-Design-Code-Test-Debug**

Programming Problem Solving Model

1. Reinterpret the Problem
2. Design a Solution
3. Code
4. Test
5. Debug
6. Evaluate & Reflect

1. Reinterpret the Problem

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

1. Reinterpret the Problem

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Input?

- Two points `(x1, y1)` and `(x2, y2)`
- Four parameters: `x1, y1, x2, y2`

1. Reinterpret the Problem

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Input?

- Two points `(x1, y1)` and `(x2, y2)`
- Four parameters: `x1, y1, x2, y2`

Output?

The distance, float

1. Reinterpret the Problem

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Input?

- Two points `(x1, y1)` and `(x2, y2)`
- Four parameters: `x1, y1, x2, y2`

Output?

The distance, float

Example - 3-4-5 triangle - `(1, 2), (4, 6) -> 5`

Incremental Development

1. Reinterpret the Problem
2. Design a Solution
3. Code
4. Test
5. Debug

First Step - Function Skeleton

```python
def distance(x1, y1, x2, y2):
    return 0.0
```

First Step - Function Skeleton

```python
def distance(x1, y1, x2, y2):
    return 0.0
```

```python
>>> distance(1, 2, 4, 6)
0.0
```

Second Step - Differences

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print("dx =", dx, "; dy = ", dy)
    return 0.0
```

Second Step - Differences

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print("dx =", dx, "; dy = ", dy)
    return 0.0
```

```
>>> print(distance(1, 2, 4, 6))
dx = 3 ; dy = 4
0.0
```

Third Step - Sum of Squares

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print("dx =", dx, "dy = ", dy)
    print("dsquared =", dsquared)
    return 0.0
```

Third Step - Sum of Squares

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print("dx =", dx, "dy = ", dy)
    print("dsquared =", dsquared)
    return 0.0
```

```
>>> print(distance(1, 2, 4, 6))
dx = 3 ; dy = 4
dsquared = 25
0.0
```

Forth Step - Square Root

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared ** 0.5
    print("dx =", dx, "dy = ", dy)
    print("dsquared =", dsquared)
    return result
```

Forth Step - Square Root

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared ** 0.5
    print("dx =", dx, "dy = ", dy)
    print("dsquared =", dsquared)
    return result
```

```
>>> print(distance(1, 2, 4, 6))
dx = 3 ; dy = 4
dsquared = 25
5.0
```

Final Version

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared ** 0.5
    return result
```

Final Version

```python
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared ** 0.5
    return result
```

```python
>>> print(distance(1, 2, 4, 6))
5.0
```

Reflection on Incremental Development

- Every time we have handled only one line of code on **Problem-Design-Code-Test-Debug** cycle
- As you gain more experience, you will find yourself managing bigger conceptual chunks
- Very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc...

Key Aspects of Incremental Development

1. Start with a working skeleton program and do **Problem-Design-Code-Test-Debug** cycles for small steps

2. Temporary variables - for intermediate values - to inspect and check easily - print them too

3. Consolidate multiple statements into compound expressions

Incremental Development Depends on Well-Preformed Test Phase

Test Phase

Programming Problem Solving Model

1. Reinterpret the Problem
2. Design a Solution
3. Code
4. Test
5. Debug
6. Evaluate & Reflect

Test Phase

Test Phase

"Surface" Goal

Validate the correctness of a program

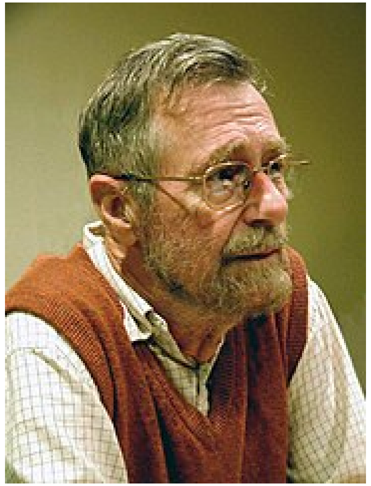
Test Phase

"Surface" Goal

Validate the correctness of a program

However...

> "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." — Edsger W. Dijkstra



Test Phase - Skeptic Mindset

Test Phase - Skeptic Mindset

Be your own Devil's Advocate

Test Phase - Skeptic Mindset

Be your own Devil's Advocate

Be a Hacker

The goal is to prove that the code is broken, not that it works

Test Phase - Skeptic Mindset

Be your own Devil's Advocate

Be a Hacker

The goal is to prove that the code is broken, not that it works

If you don't test a piece of code, you cannot trust it and you cannot depend upon it

Test Phase - Skeptic Mindset

Be your own Devil's Advocate

Be a Hacker

The goal is to prove that the code is broken, not that it works

If you don't test a piece of code, you cannot trust it and you cannot depend upon it

Re-run old tests after every new change

Learning to Test thorough Examples

Learning to Test thorough Examples

1. Quadratic Formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Learning to Test thorough Examples

1. Quadratic Formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2. FizzBuzz (3, 5)

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, Fizz Buzz, 31, 32, Fizz, 34, Buzz, Fizz, ...

Learning to Test thorough Examples

1. Quadratic Formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2. FizzBuzz (3, 5)

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, Fizz Buzz, 31, 32, Fizz, 34, Buzz, Fizz, ...

3. Find the Mode in a List

Test: "How-To"

Test Case

One "feature" or specific expected behavior

Examples

1. **Quadratic formula** - two positive roots

$$x^2 + x - 2$$

2. **FizzBuzz** - division by 3 or 5 - numbers up to 10

3. **Mode** - number with one mode - 1 2 3 2

Test Case

Test Case

Two Layers

1. Functionality / Scenario description
2. Concrete set of input-output pairs

Test Case

Two Layers

1. Functionality / Scenario description
2. Concrete set of input-output pairs

Quadratic Formula

1. Scenario: two positive roots
2. Concrete
    1. Input: `x^2 + x - 2`
    2. Output: `x1=-1.0 ; x2=2.0`

Test Case

Two Layers

1. Functionality / Scenario description
2. Concrete set of input-output pairs

Quadratic Formula

1. Scenario: two positive roots
2. Concrete
   1. Input: `x^2 + x - 2`
   2. Output: `x1=-1.0 ; x2=2.0`

FizzBuzz

1. Scenario: division by 3 or 5
2. Concrete
   1. Input: `7`
   2. Output: `1, 2, Fizz, 4, Buzz, Fizz, 7`

Test Case

Two Layers

1. Functionality / Scenario description
2. Concrete set of input-output pairs

Quadratic Formula

1. Scenario: two positive roots
2. Concrete
   1. Input: `x^2 + x - 2`
   2. Output: `x1=-1.0 ; x2=2.0`

FizzBuzz

1. Scenario: division by 3 or 5
2. Concrete
   1. Input: `7`
   2. Output: `1, 2, Fizz, 4, Buzz, Fizz, 7`

Mode

1. Scenario: numbers with one mode
2. Concrete
   1. Input: `1 2 3 2`
   2. Output: `2`

How to Come up with Test Cases?

How to Come up with Test Cases?

External / Black-box

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. ➡ Sanity Check

FizzBuzz

1. Scenario: printing numbers, `Fizz` and `Buzz`
2. Concrete
   1. Input: `7`
   2. Output: `1, 2, Fizz, 4, Buzz, Fizz, 7`

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. ➡ Trivial Cases

Mode

1. Scenario: one number
2. Concrete
    1. Input: `5`
    2. Output: `5`

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. Trivial Cases
3. Representational Cases

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. Trivial Cases
3. Representational Cases
    1. ➡ Simplest non-Trivial Cases

Mode

1. Scenario: numbers with one mode
2. Concrete
    1. Input: `1 2 2`
    2. Output: `2`

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. Trivial Cases
3. Representational Cases
    1. Simplest non-Trivial Cases
    2. ➡ General Cases

FizzBuzz

1. Scenario: divided by 3, 5 or both
2. Concrete:
    1. Input: `20`
    2. Output: `1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buz, 16, 17, Fizz, 19, Buzz`

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. Trivial Cases
3. Representational Cases
    1. Simplest non-Trivial Cases
    2. General Cases
4. ➡ Edge Cases (extreme normal)

Quadratic Formula

1. Scenario: one root
2. Concrete
    1. Input: `x^2 - 6x + 9 = (x-3)^2`
    2. Output: `x1=3.0 ; x2=3.0`

How to Come up with Test Cases?

External / Black-box

Based on the **problem** phase

1. Sanity Check
2. Trivial Cases
3. Representational Cases
   1. Simplest non-Trivial Cases
   2. General Cases
4. Edge Cases (extreme normal)
5. ➡ Corner Cases (outside normal)

Quadratic Formula

1. Scenario: no root
2. Concrete
   1. Input: `x^2 + 9`
   2. Output: `No root`

Type of Bugs - Reminder and Reflection

Type of Bugs - Reminder and Reflection

1. Syntax Error
2. Runtime Error (Exceptions)
3. Semantic/Logic Error

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

Sanity Check

- `x^2 - 3x + 2`

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

Sanity Check

- `x^2 - 3x + 2`

Simplest non-Trivial Cases

- `x^2 - 3x + 2`

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

Sanity Check

- `x^2 - 3x + 2`

Simplest non-Trivial Cases

- `x^2 - 3x + 2`

General Cases

- `+ +`: `x^2 - 10x + 18.75`
- `- -`: `x^2 + 5.8x + 5.52`
- `+ -`: `x^2 + 3.4x - 27.36`

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

Sanity Check

- `x^2 - 3x + 2`

Simplest non-Trivial Cases

- `x^2 - 3x + 2`

General Cases

- `+ +`: `x^2 - 10x + 18.75`
- `- -`: `x^2 + 5.8x + 5.52`
- `+ -`: `x^2 + 3.4x - 27.36`

Edge Cases

- b=0: `x^2 - 5`
- c=0: `x^2 - 5x`
- One root - `x^2 - 12x + 36`

How to Come up with Test Cases?

External / Black-box

Complete Test Cases (only inputs) - Quadratic Formula

Sanity Check

- `x^2 - 3x + 2`

Simplest non-Trivial Cases

- `x^2 - 3x + 2`

General Cases

- `+ +`: `x^2 - 10x + 18.75`
- `- -`: `x^2 + 5.8x + 5.52`
- `+ -`: `x^2 + 3.4x - 27.36`

Edge Cases

- b=0: `x^2 - 5`
- c=0: `x^2 - 5x`
- One root - `x^2 - 12x + 36`

Corner Cases

- a=0: `x + 6`
- no root: `x^2 + 1`

How to Come up with Test Cases?

Internal / White-box

Based on the **design** and the **code** phase

How to Come up with Test Cases?

Internal / White-box

Based on the **design** and the **code** phase

Criterion: Coverage

All possible code flows are run

How to Come up with Test Cases?

Internal / White-box

Based on the **design** and the **code** phase

Criterion: Coverage

All possible code flows are run

Look for conditionals and loops!

Example - `fizzbuzz.py`

Example - `fizzbuzz.py`

```python
def do_fizzbuzz(limit):

    values = []

    for number in range(1, limit+1):
        if number % 15 == 0:
            values.append('Fizz Buz')
        if number % 3 == 0:
            values.append('Fizz')
        elif number % 5 == 0:
            values.append('Buzz')
        else:
            values.append(str(number))


limit = int(input('Enter limit number: '))
values = do_fizzbuzz(limit)
print(', '.join(values))
```

Test Phase: In Practice

Test Phase: In Practice

When to write tests?

1. Before writing the code / function (🔥🔥🔥 **THA BEST!** 🔥🔥🔥)
2. After

Test Phase: In Practice

When to write tests?

1. Before writing the code / function (🔥🔥🔥 **THA BEST!** 🔥🔥🔥)
2. After

How to write tests?

1. Running by hand
2. `assert` (🔥🔥🔥 **THA BEST!** 🔥🔥🔥)

Test Phase

Wrap-up + Q&A

(this is not the end yet)

Evaluate & Reflect Phase

Programming Problem Solving Model

1. Reinterpret the Problem
2. Design a Solution
3. Code
4. Test
5. Debug
6. Evaluate & Reflect

Evaluate & Reflect Phase

Evaluate & Reflect Phase

Evaluation

Outcome - Code

Evaluate & Reflect Phase

Evaluation

Outcome - Code

Reflection

Process - Problem Solving

Evaluate & Reflect Phase

Evaluation

Outcome - Code

Reflection

Process - Problem Solving

We will focus today on **Evaluation**

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

We will focus today on **Functionality**, which depends on the *Test* phase

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

We will focus today on **Functionality**, which depends on the *Test* phase

Functionality

The code ...

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

We will focus today on **Functionality**, which depends on the *Test* phase

Functionality

The code ...

1. ... is running and not crashing

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

We will focus today on **Functionality**, which depends on the *Test* phase

Functionality

The code ...

1. ... is running and not crashing
2. ... solves the problem (meets the task requirements)

Evaluation Criteria

1. Functionality
2. Design and code
3. Readability, style & documentation

We will focus today on **Functionality**, which depends on the *Test* phase

Functionality

The code ...

1. ... is running and not crashing
2. ... solves the problem (meets the task requirements)
3. ... is robust (for *edge* and *corner* test cases)

Evaluation - Functionality

Example `quadratic.py`

Evaluation - Functionality

Example `quadratic.py`

The program works well for the majority of the inputs. The program gives correct results for the *representational cases* (two different roots, all the three combinations of root signs).

Evaluation - Functionality

Example `quadratic.py`

The program works well for the majority of the inputs. The program gives correct results for the *representational cases* (two different roots, all the three combinations of root signs).

The programs also works well for the *edge* cases (`b=0`, `c=0`, one root).

Evaluation - Functionality

Example `quadratic.py`

The program works well for the majority of the inputs. The program gives correct results for the *representational cases* (two different roots, all the three combinations of root signs).

The programs also works well for the *edge* cases (`b=0`, `c=0`, one root).

However, the program fails on for *corner* cases:

Evaluation - Functionality

Example `quadratic.py`

The program works well for the majority of the inputs. The program gives correct results for the *representational cases* (two different roots, all the three combinations of root signs).

The programs also works well for the *edge* cases (`b=0`, `c=0`, one root).

However, the program fails on for *corner* cases:

1. For the input of `a=0`, the program raises `ZeroDivisionError` exception, because there is no check for that.

Evaluation - Functionality

Example `quadratic.py`

The program works well for the majority of the inputs. The program gives correct results for the *representational cases* (two different roots, all the three combinations of root signs).

The programs also works well for the *edge* cases (`b=0`, `c=0`, one root).

However, the program fails on for *corner* cases:

1. For the input of `a=0`, the program raises `ZeroDivisionError` exception, because there is no check for that.

2. for a equation without (real) roots (e.g. `x^2 + 1`), the program raises `ValueError: math domain error` because the value beneath the square root is negative.

Wrap-up

Problem Solving using Python - Week 4

Incremental Development, Test, Evaluation

NETFLIX

SUITS

SUITS

SUITS

Going to trial

is already loosing

SUITS

Going to ~~trial~~ DEBUGGING is already loosing

How to Avoid Debugging?

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

Problem

Solution

How to Avoid Debugging?

1. ➡ Write **tests** at the Problem phase (e.g. using `asserts`)
2. Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

Problem

Solution

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
   1. Start small
   2. Keep it working

Problem

Solution

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

Problem 📝

Solution

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
   1. Start small
   2. Keep it working

Problem 📝 📝 **Solution**

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
   1. Start small
   2. Keep it working

Problem 📝 📝 📝 **Solution**

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

Problem 📝 📝 📝 📝 **Solution**

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
   1. Start small
   2. Keep it working

Problem 📝 📝 📝 📝 📝 **Solution**

`assert`

How to Avoid Debugging?

1. Write **tests** at the Problem phase (e.g. using `asserts`)
2. ➡ Use **incremental development** = Get something working and keep it working
    1. Start small
    2. Keep it working

Problem 📝 📝 📝 📝 📝 **Solution**

`assert`

🎉 🎊

Q&A

Problem Solving using Python - Week 4

Incremental Development, Test, Evaluation