Extended Introduction to Computer Science CS1001.py

Lecture 10, part A: Interim Summary; Testing; Coding Style

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University

Fall Semester 2017
http://tau-cs1001-py.wikidot.com

Lecture 1-9: Highlights

What have we learned in the first 5 weeks of the course?

(or: What did you learn in school today, Dear little boy of mine?)

- Quick intro to Python
  - types, functions, loops, conditionals, lists, tuples…
  - Python's memory model

- Natural numbers representation in binary and other bases
- Integer exponentiation: naïve (slow) vs. iterated squaring (fast)
- Basic algorithms: binary search, selection sort, merge
- Complexity (theoretic analysis, O(…) notation, and actual measurements)
- Lambda expressions (anonymous functions)
- High order functions
- Floating point representation
- Finding roots of real values functions (binary search, Newton-Raphson)
- Randomness in the context of computing. Pseudo random generation.

Lecture 10 to +∞

It's time for you to recap on everything, see that you are ready for what's coming next.

The rest of the course is deals mostly with the following topics:

- Recursion
- Algorithms of large integers (primality testing, cryptography)
- Object oriented programming (OOP)
- Data structures (linked lists, trees, hash tables)
- Strings and Text (string matching, text compression)
- Image processing
- Error correction codes
- The halting problem and computationally hard problems

Lecture 10.3

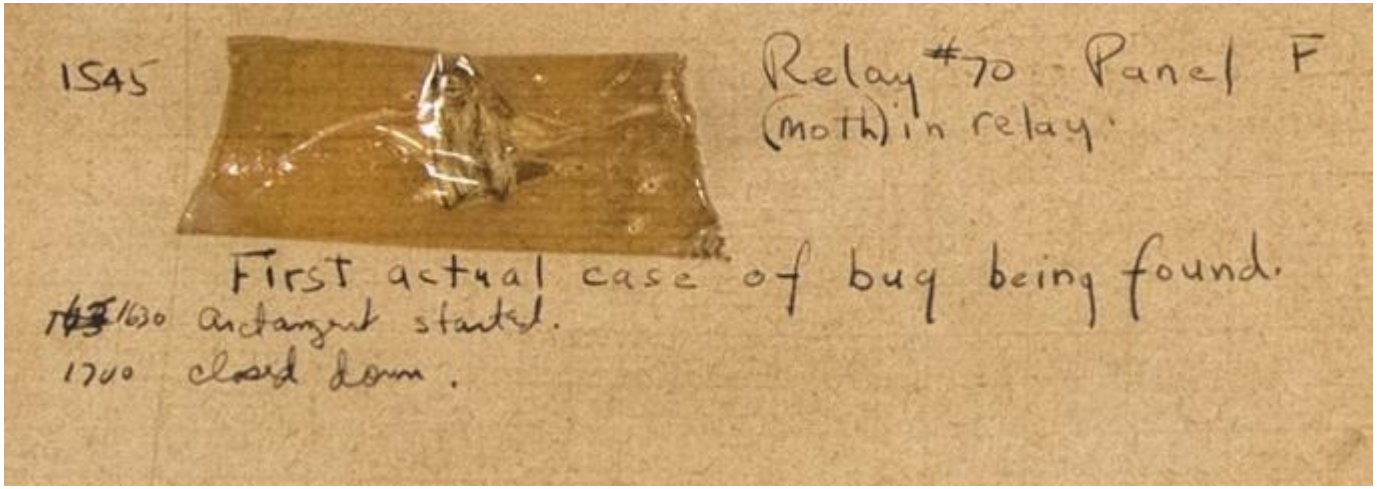
It is the time to mention 2 issues related to software development:

- Testing
- Styling

- Both subject will be over-simplified. We mostly want to expose you to the basic ideas and be aware of the issues.

Bugs

A computer bug is an error that causes a computer to produce an incorrect or unexpected result.



1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~143~~1630 Antangent started.

1700 closed down.

(see https://www.wired.com/2009/09/happy-birthday-computer-bugs/)

Types of errors:

- Syntax errors
- Run-time errors
- Logical errors

Types of Bugs

- Syntax errors, e.g.:

  e.g. incorrect indentation, missing elements (like ':').

  Easiest to find, as the interpreter yells at us:

  ```
  >>> def t1()
  SyntaxError: invalid syntax
  ```

- Run-time errors

  e.g. division by 0, illegal access to memory (lst[len(lst)])

  Often go unnoticed, until they actually happen:

  ```
  >>> 4/0
  ZeroDivisionError: int division or modulo by zero
  ```

- Logical / algorithmic errors (this code does not crash but is incorrect)

  Usually hardest to find.

  Surely IDLE will not find them.

```
#This program computes the average of 100 numbers
s = 0
i = 1
while i <= 100:
    next_num = float(input("Enter a number"))
    s = s + next_num
    i = i + 1
print(s/i)
```

Debugging / Testing

- Testing is the process of executing a program, with the **intent of finding errors**.
- Debugging is the process of locating the origins of these errors

  recall that original bugs were real bugs within the circuitry, causing short circuits
- This is really a whole world, an expertise.
- It is argued by some, that programmers should not do the testing, i.e., these functionalities should be separated.
- Others support a TDD approach (test driven development)
- The common testing approach combines both programmer testing with QA testing.

Testing categories

Just to mention the names of the important ones:

- Static vs. dynamic Testing
- Black-Box vs. white-box Testing
- Top-down vs. bottom-up Testing

- We will not elaborate on these in our course, though. You will probably meet them again in more advanced software courses, esp. in elective software engineering courses.

Tips!!

You WILL (and probably already did) use basic testing while writing code for your HW (assuming it was indeed you who wrote it).

Our recommendions for YOU to test your code in this course:

1. Diagnostic printouts

   Strategically place print() statements to enable following information flow.

2. “Divide and conquer”

   Unit test

   Test a single unit (e.g. function) at a time.

3. Mocking

   Mocks replace real objects and simulate their behavior in an expected / fixed way. They enable isolating the behavior of the tested object.

Example from HW2

```
def sum_divisors(n):
    return ...
```

Mock by naïve, not O($\sqrt{n}$) implementation:
for i in range(n)…

and test is_perfect

```
def is_perfect(n):
    return True/False
```

Replace, for example, by:
- Always return True
- return True for 3, 16 and 74.

to test cnt_perfect_numbers

```
def cnt_perfect_numbers(limit):
    return ...
```

IDLE Debugger (for reference only)

Most programming environments provide debuggers, and so does IDLE.

But, we do not use it in this course.

Debuggers enable a step-by-step tracking of the program’s state, i.e. values in the memory.

Code Style

Code Style

- Writing “nice” code is sometimes considered an art.
- Recall: beauty is in the eyes of the beholder…
- However there are some common practices, which are good to be aware of and follow.

- PEP8 – Python Style Guide

  http://legacy.python.org/dev/peps/pep-0008/

  (PEP = Python Enhancement Proposals)

- Next are some highlights from PEP8 – for self reading.
- Except for the next slide, consider these as recommendations.

  We will not always fully comply ourselves (practice what you preach?).

Two Important Musts

- Give meaningful names to variables and functions

Bad examples - Bad meaning:

var

bla

something

tmp (except for a temporary auxiliary variable)

x,y (except for a real number, point coordinates, etc.)

n,m (except for some integer such as size)

i,j (except for an index)

Bad example - too much meaning:

the_name_of_my_cs_intro_lecturer_that_is_not_Amir = “Benny”

- Consistency in style is important, within a project, and even more so, within one module or function.

Key Styling features - spaces

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Good: spam(ham[1], {eggs: 2})
Bad: spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon:

```
Good: if x == 4: print(x, y); x, y = y, x
Bad: if x == 4 : print(x , y) ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Good: spam(1)
Bad: spam  (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Good: dict['key'] = list[index]
Bad: dict ['key'] = list [index]
```

Key Styling features - spaces

Always surround these binary operators with a single space on either side:

assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Good:

```
i = i+1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Bad:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Key Styling Features - Naming Styles

The following naming styles are commonly distinguished:

- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (aka CamelCase - so named because of the bumpy look of its letters
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

Be consistent !

Also, avoid the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.