Extended Introduction to Computer Science
CS1001.py

Lecture 8 part B: testing; coding style

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Yael Baran

School of Computer Science
Tel-Aviv University
Spring Semester 2015
http://tau-cs1001-py.wikidot.com
Lecture 1-8: Highlights

What have we learned in the first month of the course?

• Quick intro to Python
  • types, functions, loops, conditionals, lists, tuples...
  • Python's memory model
  • More on functions:
    • anonymous functions (lambda expressions)
    • high order functions

• Integer representation in binary and other bases
• Integer exponentiation (naïve vs. iterated squaring)
• Basic algorithms: search, sort, merge
• Complexity (real times and theoretic analysis with $O(\ldots)$)
• Floating point representation
  • roots of real values functions (binary search, Newton-Raphson)
Lecture 9 - ∞

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 more or less organized in this structure:

• Recursion
• Issues in Number Theory (including intro to cryptography)
• Data Structures (classes, trees, linked lists, hash tables)
• Strings and Text
• miscellaneous
  • Image processing
  • Error correction
  • The halting problem and computationally hard problems
Lecture 8½

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

• testing
• Styling

• Both will be over-simplified. We just want to expose you to the basic ideas.
Bug

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

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)])
May go unnoticed, until they happen:

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

• Logical / algorithmic errors

Usually hardest to find.
We cannot expect IDLE to find them.

```python
#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. Original bugs were real bugs found within the circuitry and caused 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** (testing driven development)

• Anyhow, you WILL use (and probably already did) basic testing for your HW solutions.
Testing categories

Just to mention the important ones:

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

We will not explain them. You will probably hear about this again in more advanced software courses.
Tips!!

How do we recommend for YOU to test your functions in this course?

1. Diagnostic printouts
   Strategically place print() statements to follow information flow

2. “Divide and conquer”
   Unit test
   Test a single unit (e.g. function) at a time

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 ...

Replace by naïve implementation:
    for i in range(n)...

to test is_finite

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

to test cnt_finite

def is_finite(n):
    return True/False

def cnt_finite(limit):
    return ...
Most programming environments provide debuggers, and so does IDLE.

We do not use it in this course.
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.

• 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.
Two important musts

• Give meaningful names to variables and functions

Bad examples - no meaning:

```javascript
var
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:

```javascript
the_name_of_my_cs_intro_lecturer_that_is_not_amir = “Benny”
```

• Consistency in style is important, within a project, and even more within one module or function.
Key Styling features - spaces

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

  Yes: spam(ham[1], {eggs: 2})
  No: spam( ham[ 1 ], { eggs: 2 } )

- Immediately before a comma, semicolon, or colon:

  Yes: if x == 4: print(x, y); x, y = y, x
  No: if x == 4 : print(x , y) ; x , y = y , x

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

  Yes: spam(1)
  No: spam (1)

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

  Yes: dict['key'] = list[index]
  No: 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.

Yes:
\[
\begin{align*}
  i &= i + 1 \\
  \text{submitted} &= 1 \\
  x &= x \cdot 2 - 1 \\
  \text{hypot}^2 &= x \cdot x + y \cdot y \\
  c &= (a + b) \cdot (a - b)
\end{align*}
\]

No:
\[
\begin{align*}
  i &= i + 1 \\
  \text{submitted} &= 1 \\
  x &= x \cdot 2 - 1 \\
  \text{hypot}^2 &= x \cdot x + y \cdot y \\
  c &= (a + b) \cdot (a - b)
\end{align*}
\]
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 'l' (uppercase letter eye) as single character variable names.