# Extended Introduction to Computer Science
# CS1001.py

## Chapter A Error Types
## Lecture 6b Tips for “Good” Code

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University

Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

# מבנה ונושאי הקורס (באדום - חומר שירד בשל קיצור הסמסטר)

| פרק | נושאים מתוכננים |
|---|---|
| A. יסודות פייתון | • תכנות בסיסי: טיפוסי ערכים, משתנים, משפטי תנאי, לולאות, פונקציות, מודל הזיכרון<br>• נושאים נוספים: דקדוקים פורמליים ותהליך הפירוש של פייתון, פונקציות למבדא, ופונקציות סדר גבוה, אקראיות ושימושיה, סוגי שגיאות (תחביר, זמן ריצה), סגנון תכנות "נכון" |
| B. ייצוג טיפוסי מידע | • ייצוג שלמים בשיטה הבינארית<br>• ייצוג מספרים עם נקודה עשרונית בשיטת floating point<br>• ייצוג תווים (ASCII, Unicode) |
| C. אלגוריתמים בסיסיים וסיבוכיות | • חיפוש בינארי, מיון בחירה, מיזוג רשימות ממוינות<br>• סיבוכיות ו- O notation |
| D. חישוב נומרי | • מציאת שורש של פונקציה ממשית רציפה בשיטת החציה ~~בעבר: שיטת ניוטון-רפסון~~, חישוב נגזרות ואינטגרלים, קירוב ל $\pi$ |
| E. רקורסיה | • עצרת, פיבונאצ'י, חיפוש בינארי, מיון מהיר, מיון מיזוג, ממואיזציה, דוגמאות נוספות |
| F. נושאים בתורת המספרים | • העלאה בחזקה טבעית בשיטת Iterated squaring<br>• בדיקת ראשוניות הסתברותית (המשפט הקטן של פרמה)<br>• פרוטוקול Diffie-Hellman להחלפת מפתח סודי<br>• מחלק משותף מקסימלי (GCD) |
| G. תכנות מונחה עצמים (OOP) ומבני נתונים | • מחלקות, שדות ומתודות<br>• רשימות מקושרות והשוואה לרשימות של פייתון<br>• עצי חיפוש בינאריים<br>• טבלאות hash<br>• זרמים (streams) ופונקציות גנרטור |
| H. טקסט | • אלגוריתם CYK ~~בעבר: אלגוריתם קארפ-רבין~~<br>• דחיסת האפמן, דחיסת למפל זיו |
| I. ייצוג ועיבוד תמונה | • ייצוג תמונה דיגיטלית, ניקוי רעש (ממוצע וחציון מקומי), נושאים נוספים לפי הזמן |
| J. קודים לגילוי ולתיקון שגיאות | • ספרת ביקורת, קוד חזרה, ביט זוגיות, מרחק האמינג, קוד האמינג |

You are here

# The Three C's* of Good Programming

- Correctness:
  - is it correct?
  - what are the special cases?

- Complexity:
  - is it efficient enough?
  - can we improve?

- Clarity
  - can we write it simpler or "nicer" at no significant cost?
  - is the code easy to modify / extend?

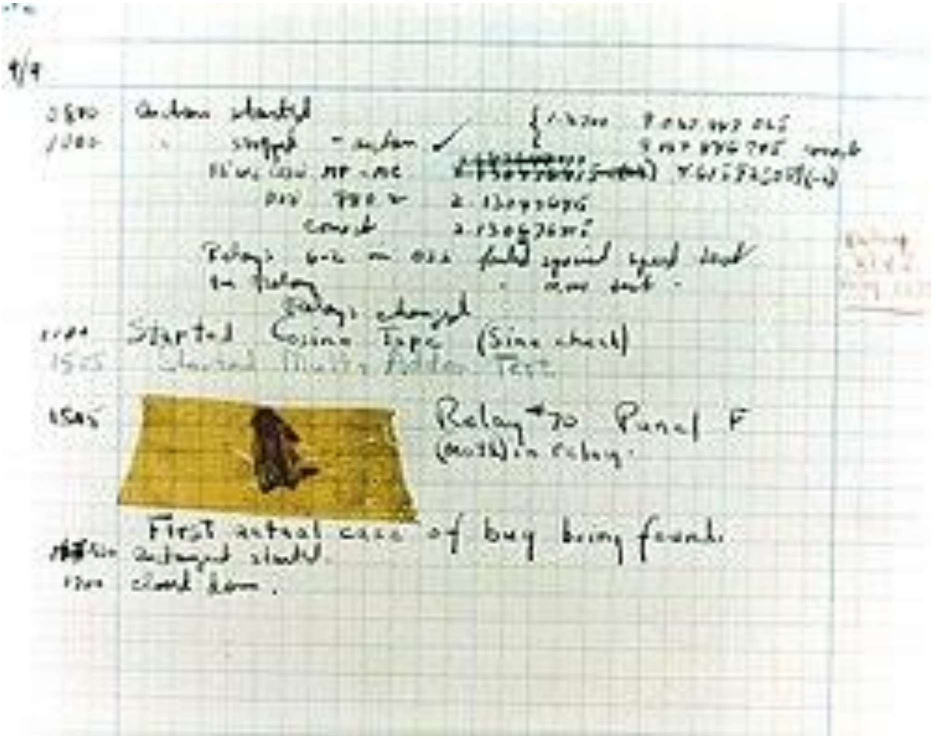
* Slide prepared by AR, inspired by a recent (2020) political interview in Israel

# Bugs in Programs

- A bug is an error in the program that causes incorrect or unexpected behavior

- The term originates from a real bug that caused short circuits in hardware (see this article)

- Debugging aims at removing such errors. One possible approach is testing: executing a program, with the intent of finding errors



9/9

0800 Anctan started

1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 795 correct

13" UC (032) MP-MC ~~[illegible]~~) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test
In relay " " test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multi Adder Test.

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

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Image from Wikipedia.
A page from the Harvard Mark II electromechanical computer's log, featuring a dead moth that was removed from the device (1947).

# Types of Errors

- Syntax errors
  - Commands that do not follow the syntax of a programming language (as defined by its grammar), thus interpretation fails
  - Discovered during interpretation

- Run-time errors
  - Program is syntactically correct, but some operations cannot be completed and program execution is terminated abruptly
  - Discovered during execution

- Semantic errors
  - Program runs and completes, but produces incorrect results
  - Hardest to discover

# Syntax Error Examples

```
3 = x
SyntaxError: can't assign to literal
```

```
def f()
   return 3
SyntaxError: invalid syntax
```

```
if 3<4:
else:
SyntaxError: invalid syntax
```

# Runtime Error Examples

```
4/0
Traceback (most recent call last):
    ...
    4/0
ZeroDivisionError: division by zero
```

```
L = [1,2,3]
L[3]
Traceback (most recent call last):
    ...
    L[3]
IndexError: list index out of range
```

```
a=4
print(A)
Traceback (most recent call last):
    ...
    print(A)
NameError: name 'A' is not defined
```

# Semantic Error Example

```
def average10():
    """ compute the average of 10 numbers """
    s = 0
    i = 1
    while i <= 10:
        next_num = float(input("Enter next number"))
        s += next_num
        i += 1
    return s/i
```

# Tips!!

1. Diagnostic printouts

   Strategically place print() statements to track information flow

2. Unit test ("divide and conquer")

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

   Rely on other units' correctness

3. Python tutor and similar tools

4. Debugger

   Most IDE's (including IDLE) provide debuggers – tools that ease tracking a program's execution, step by step.

   We do not use debuggers in this course (but you are welcome to try it).

# The Three C's* of Good Programming

- Correctness:
  - is it correct?
  - what are the special cases?

- Complexity
  - is it efficient enough?
  - can we improve?

- Clarity
  - can we write it simpler or "nicer" at no significant cost?
  - is the code easy to modify / extend?

* Slide prepared by AR, inspired by a recent (2020) political interview in Israel

# Writing “Good” Code

- Writing “good” 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. These often have significant affect on correctness and cost as well.

- We will explore some of these via examples.
- Note: all examples are correct in the sense of input-output relation.

# Example 1a

- Can you figure out what this code is all about?

```
def f(x):
   if x%4 == 0:
      if x%100 == 0:
         if x%400 == 0:
            return True
         else:
              return False
      else:
         return True
   else:
      return False
```

# Example 1b

- And now?

```
def is_leap(year):
    if year%4 == 0:
        if year%100 == 0:
            if year%400 == 0:
                return True
            else:
                   return False
        else:
            return True
    else:
        return False
```

# Example 1c

- And how about this one?

```
def solve(a,b,c):
    d = (b**2) - (4*a*c)   # the discriminant

    sol1 = (-b-(d**0.5))/(2*a)
    sol2 = (-b+(d**0.5))/(2*a)

    return sol1, sol2
```

# 1. Meaningful Names

- Meaningful names to both variables and functions should make your code more self-explanatory
    - For example, a general counter should be named: `cnt/count/counter/cntr/`…
    - A specific counter can be named, e.g.: `cnt_days, cnt_zeros`, etc.
    - Use indices (e.g. `cnt1, cnt2, cnt3`) to indicate variables with similar “roles”
    - Context should be taken into consideration (as in the last example)

- Bad names:
    - ✗ `variable`, `something`, `func`
    - ✗ `tmp` (except for a temporary auxiliary variable)
    - ✗ `l`, `I`, `O`, `o`
    - ✗ `the_name_of_my_cs_intro_lecturer = "Amir"`

- Names often assigned for numbers:
    - `x`, `y`, `z` - for real numbers, point coordinates, etc.
    - `n`, `m`, `k` - for integers
    - `i`, `j`, `k` - for indices

# Naming Conventions

- The following naming styles are commonly recommended in Python:

  - `lower_case_with_underscore` - used for variables and functions names
  - `CapitalizedWords` – (aka CamelCase) often used for classes (later in this course)
  - `UPPER_CASE_WITH_UNDERSCORE` - used for program constants

- Consistency is more important than choosing any specific convention (which may occasionally even turn into an ideological and emotional debate)

# Example 2a

```
def count_positives(lst):
    ''' count numbers > 0 in lst '''
    positives = []
    for x in lst:
        if x > 0:
            positives.append(x)
    return len(positives)
```

vs.

```
def count_positives(lst):
    ''' count numbers > 0 in lst '''
    cnt = 0
    for x in lst:
        if x > 0:
            cnt += 1
    return cnt
```

# Example 2b

```
def days_in_month(month):
   ''' how many days in month (1=Jan,2=Feb,…,12=Dec) '''
   if month == 2:
      return 28
   elif month == 1 or month == 3 or month == 5 or month == 7    \
                   or month == 8 or month == 10 or month == 12:
      return 31
   return 30
```

vs.

```
def days_in_month(month):
   ''' how many days in month (1=Jan,2=Feb,…,12=Dec) '''
   long_months = [1, 3, 5, 7, 8, 10, 12]
   if month == 2:
      return 28
   elif month in long_months:
      return 31
   return 30
```

# 2. Memory Usage

- Often storing data in memory can save time or make the code clearer

- However, use memory efficiently: large memory consumption is likely to slow the execution down, or even crash it
    - Don’t try at home:

```
L = [i for i in range(10**10)]
```

# Example 3

```
def power2(exp):
    ''' return 2^exp '''
    if exp == 0:
        return 1

    res = 2
    for i in range(1, exp):
        res *= 2


    return res
```

vs.

```
def power2(exp):
    ''' return 2^exp '''
    res = 1

    for i in range(0, exp):
        res *= 2


    return res
```

# 3. Special Case Treatment

- Handling special cases separately often (but not always) makes the code complicated and less readable

- Avoid unnecessary separation of special cases, except when you have a good reason

# Example 4

```
def indices_of_odds(lst):
    '''
    print indices of odd numbers
    '''
    i = 0

    while i < len(lst):
        if lst[i]%2 == 1:
            print(i)
        i += 1

    return None
```

vs.

```
def indices_of_odds(lst):
    '''
    print indices of odd numbers
    '''
    i = 0

    for num in lst:
        if num%2 == 1:
            print(i)
        i += 1

    return None
```

vs.

```
def indices_of_odds(lst):
    '''
    print indices of odd numbers
    '''
    for i in range(len(lst)):
        if lst[i]%2 == 1:
            print(i)

    return None
```

# 4. Iteration Structure

- Choose the simplest and most appropriate loop structure
    - while vs. for
    - direct vs. indirect
    - nested vs. flat

# Example 5b

```
def control_digit(ID):

    total = 0
    for i in range(8):
        if i % 2 == 0:
            total += int(ID[i])
        else:
            if int(ID[i]) < 5:
                total += 2*int(ID[i])
            else:
                total += (2*int(ID[i]) % 10) + 1

    total = total % 10
    check_digit = (10 - total) % 10

    return str(check_digit)
```

# Example 5b

```
def control_digit(ID):

    total = 0
    for i in range(8):
        digit = int(ID[i])
        if i % 2 == 0:
            total += digit
        else:
            if digit < 5:
                total += 2*digit
            else:
                total += (2*digit % 10) + 1

    total = total % 10
    check_digit = (10 - total) % 10

    return str(check_digit)
```

# 5. Code Duplication

- Avoid duplicating the same computation multiple times- this may have negative effects on the time / memory costs of your code

- Store useful data in variables, useful code in functions

# Example 6

```
def sign(num):
    ''' sign of a number '''
    if num == 0:
        return 0
    else:
        if num > 0:
            return 1
        else:
            if num < 0:
                return -1
```

vs.

```
def sign(num):
    ''' sign of a number '''
    if num == 0:
        return 0
    elif num > 0:
        return 1
    else:
        return -1
```

vs.

```
def sign(num):
    ''' sign of a number '''
    return 2*(num > 0) - 1 + (num == 0)
```

# 6. Simplicity

- Keep it simple (both visually and logically), unless you know what you’re doing (and why)

# Example 7a

```
def reverse_lst(lst):
    return lst[::-1]
```

vs.

```
def reverse_lst(lst):
    rev = []
    n = len(lst)

    for i in range(n):
        rev.append(L[n-i-1])

    return rev
```

# Example 7b

```
def second_largest(lst):
   ''' computes second largest number in list '''
   lst.sort()
   return lst[-2]
```

vs.

```
def second_largest(lst):
   ''' computes second largest number in list '''
   max1 = max2 = lst[0]

   for num in lst:
      if num > max2:
         if num > max1:
            max2 = max1
            max1 = num
         else:
            max2 = num

   return max2
```

# 7. “Under the Hood”

- Python is a powerful language with many built-in shortcuts, but with great power comes great responsibility

# 8. Comments

- Comments (even good ones) do not excuse unclear code

- Before writing a comment, consider clarifying the code itself

# Some Funny Comments

```
# When I wrote this, only God and I understood what I was doing
# Now, God only knows
```

```
# I dedicate all this code, all my work, to my wife, Darlene,
# who will have to support me and our three children and the
# dog once it gets released into the public.
```

```
# I am not responsible of this code.
# They made me write it, against my will.
```

```
# drunk. fix later
```

```
# Magic. Do not touch.
```

```
# I am not sure if we need this, but too scared to delete.
```

```
# Dear future me. Please forgive me.
# I can't even begin to express how sorry I am.
```

```
# no comments for you!
# it was hard to write so it should be hard to read
```

# Python Styling Conventions (for reference only)

- Recall: beauty is in the eyes of the beholder…

- PEP8 – Python Style Guide

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

  (PEP = Python Enhancement Proposals)

- Online style checker: https://www.codewof.co.nz/style/python3/

- We do not always follow these conventions ourselves…

# Additional Styling Features - Spaces

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Yes: func(var[1], {key: 2})
No:  func( var[ 1 ], { key: 2 } )
```

- Immediately before a comma, semicolon, or colon:

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

- 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]
```

# Additional Styling Features - Spaces

- Always surround the following binary operators with a single space on either side:
  assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, etc.),
  Booleans (and, or, not), in, is.

- 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:
i = i+1
cnt += 1
x = x*2 - 1
res = x*x + y*y
```

```
No:
i=i+1
cnt +=1
x = x * 2 - 1
res = x * x + y * y
```