

CMPT 120

Intro to CS & Programming I

WEEK 2 (Jan. 13-17)

— *Jérémie O. Lumbroso* —

Lecture 5:
Elementary Data Types in Python

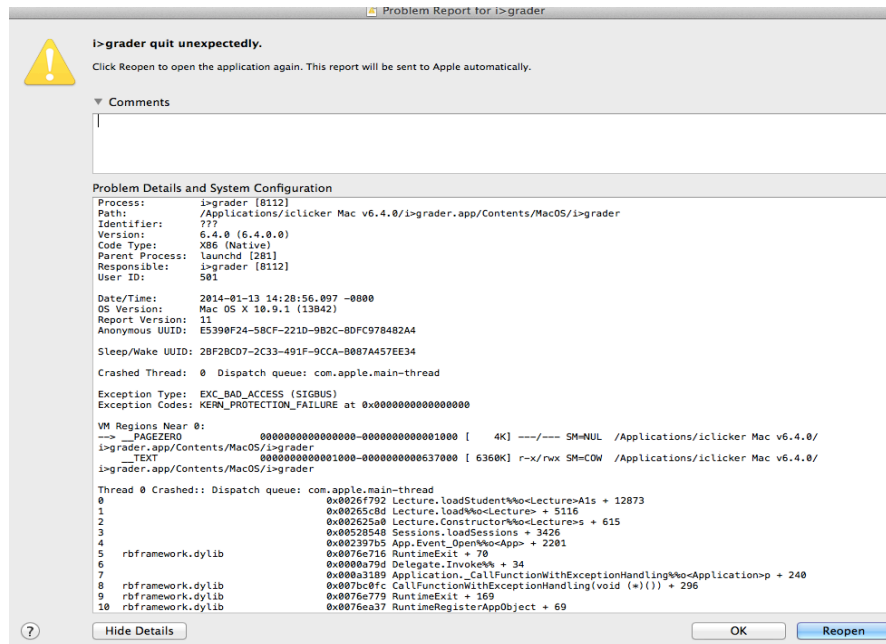
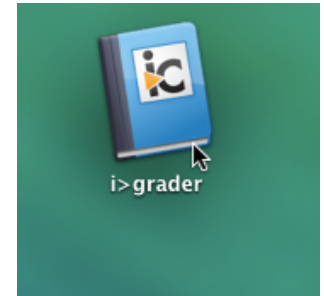
<http://www.sfu.ca/~jlumbros/Courses/CMPT120/>

Before we start...

STORY OF HOW YOUR PROF SAVED THE DAY

iClicker tribulations

- After the second lecture, I wanted to go over the iClicker results
- Then the following happened



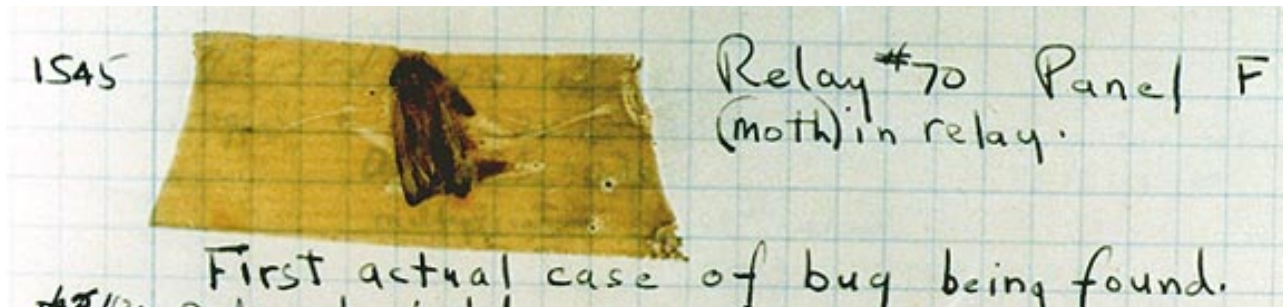
iClicker Bug??

- iClicker stores lots of **timing information** (when you cast a vote, when I stop the timer, etc.)
- This time: expressed as **seconds and milliseconds**
- Supposed to be separated by a **period**

27.654

- But with my regional settings (France) separated by a **comma**

27,654



iClicker Bug!!

- iClicker stores information as CSV (comma separated values)

```
#000DE2EF, "", 0, A, 0, 20.033, 4, A, 17.167, B, 0, 25.750, 1, B, 25.750, C, 0, 20.717, 3, C, 9.850, B, 0, 29.867, 1, B, 29.867, B, 0, 5.900, 1, B, 5.900,
```

- But on my computer, the files looked like this

```
#000DE2EF, "", 0, A, 0, 20,033, 4, A, 17,167, B, 0, 25,750, 1, B, 25,750, C, 0, 20,717, 3, C, 9,850, B, 0, 29,867, 1, B, 29,867, B, 0, 5,900, 1, B, 5,900,
```

- This typo happened twice, for each question, for every student, for every lecture, so **well over 10,000 times**
- What to do?
- Pretend to grade you on iClicker responses and then make up a value?

iClicker Bug Solution!

- No! Write a [Python] script!

```
import fileinput      # Input/output module
import re             # Regular expression module

def correctLine(line):
    return re.sub(r"([0-9]{3})(?!,[0-9]{3})",r".\1", line)

for line in fileinput.input():    # Read lines from prompt
    if len(line) != 0 and line[0] == '#':
        print correctLine(line),
    else:
        print line,      # Print line without a linebreak
```

Manipulating elementary values in Python

BASIC DATA TYPES AND OPERATIONS

(Like You) a Value Has a Type

- The data expressed in Python can be of any of **different type**
- This type determines **how the data can be manipulated**: what operations? what comparisons? etc.
- Variables can be assigned any value
- Because there is no “**declaration**” in Python (when you have to say you are going to use a variable name), **types are guessed** by Python

TYPE	EXAMPLES	NOTES
int	76, 4, 1093847, 2384	arbitrary precision (can do very big numbers!)
float	5.4, 10.7632, 0.33333	only 16 digits of accuracy
string	"Hey!", "p&j sandwich"	
tuple	(1, 9, 2, 7), (12, 45)	can't be modified
list	[4, 5, 193, 24], [3, 13]	

Try Some Calculations



Using your computer (or your neighbors) try these expressions out, and give the correct answer.

```
>>> print 3 - 5
```

- A 2 B -2 C 35 D 53 E 1

```
>>> print 10/3
```

- A 3.3333333 B 10/3 C 3 D 4 E 1

```
>>> print 10/3.
```

- A 3.3333333 B 10/3 C 3 D 4 E 1

```
>>> print 10./3
```

- A 3.3333333 B 10/3 C 0 D 4 E 1

```
>>> print 10*10*10*10+100+1
```

- A 10101 B 1010101001 C 100001001 D 111 E 11111111

Try Some More Calculations



Using your computer (or your neighbors) try these expressions out, and give the correct answer.

```
>>> print 10%3
```

- A 3.3333333 B 10/3 C 3 D 4 E I

```
>>> print (4+5) * (5+4) + (3*2) + (1*2)
```

- A 89 B 93 C 72 D 127 E ERROR

```
>>> print (1+4) (3+4)
```

- A (5)(7) B 57 C 35 D 53 E ERROR

```
>>> print 100^1^10^100^1^10
```

- A III B 100 C 0 D 1000000000 E 100110100110

```
>>> print 100**1**10**100**1**10*2
```

- A 100 B 222 C 200 D 0 E 200220200220

Operations on Numbers

- Python operators $+$, $-$, $*$, $/$ work as you would expect with mathematical precedence
- Like in math, parentheses (and to) change precedence (order of operations)
- **Division on integers** is the **quotient** of the whole division (= rounded down)
- **Division of floats** gives a float ($1 ./ 3. = 0.3333\dots$)
- When **mixing floats and integers**, Python converts everything to floats
- Remainder/modulo is $\%$, exponentiation is $**$, and the $^$ operator is severely messed up!

Calculations with Strings



Using your computer (or your neighbors) try these expressions out, and give the correct answer.

```
>>> print "abc" + "def"
```

- A abcdef B fhj C adbecf D fhjfhj E ERROR

```
>>> print "abc" * 3
```

- A abcabcabc B aaabbbccc C dgj D dgjdgjdgj E ERROR

```
>>> print "abc" + 3
```

- A abcabcabc B aaabbbccc C def D abc3 E ERROR

```
>>> print "120" * "3"
```

- A 360 B 1203 C 120120120 D 111222000 E ERROR

```
>>> print "120" * 3
```

- A 360 B 1203 C 120120120 D 111222000 E ERROR

Operations on Strings

- Strings are **contained in quotes** `"a string"`
- **Concatenation** (appending two strings) can be done with the `+` operator
- Strings can be repeated with `*`
- Anything in quotes (including a number) is a string; for example, `"120"` is not a number
- It is **not possible to concatenate a string with something that is not a string**, later we will see
 - type conversion (to go from int to string, for instance)
 - string formatting
- Get the length of string `mystr` with `len(mystr)`

Slicing String



```
>>> myvar = "my string"
```

```
>>> print myvar
```

A myvar B "my string" C my string D "myvar" E ERROR

```
>>> print my string
```

A myvar B "my string" C my string D mystring E ERROR

```
>>> print myvar[1]
```

A m B y C my D string E ERROR

```
>>> print myvar[0:2]
```

A my B y s C my string D m E ERROR

```
>>> print myvar[3:]
```

A my B string C s D my s E ERROR

Slicing Strings

- String can be **sliced**: a character or range of character can be accessed using the following syntax
 - `mystring[k]` gives character #k of my string
 - `mystring[a:b]` gives range from character #a to #b
 - `mystring[a:]` gives range from character #a to end
 - `mystring[:b]` gives range from beginning to #a
- **Important:** strings (and everything else in Python) are indexed in 0; this means that the first character of a `mystring` is `mystring[0]` **not** `mystring[1]`



How to program an `if` in Python, and how to iterate over a range of ints

ELEMENTARY CONTROL STRUCTURES

if Statement in Python

- Keywords are `if`, `elif` (else if) and `else`
- Only the `if` is mandatory
- Syntax is `if <condition>:`

```
if True:  
    print "This will always print!"
```

```
numToGuess = 17  
if numToGuess > 10 or numToGuess < 0:  
    print "Your number is not between 1 and 10"  
elif numToGuess == 5:  
    print "Your number is 5"  
else:  
    print "Your number is between 1 and 10"  
    print "But it's not 5 :-(
```

Blocks in Python



- Blocks in Python have the following characteristics
 - the line before the block ends with a colon :
 - the block must be indented, and always indented the same

```
if someCondition:  
    print "someCondition was verified"  
    print "this is good"  
else:  
    print "someCondition: not verified"
```

Correct

```
if someCondition  
    print "someCondition was verified"  
    print "this is good"  
else  
    print "someCondition: not verified"
```

Wrong: no colons


```
if someCondition:  
    print "someCondition was verified"  
    print "this is good"  
else:  
    print "someCondition: not verified"
```

Correct: indentation same within each block

```
if someCondition:  
    print "someCondition was verified"  
    print "this is good"  
else:  
    print "someCondition: not verified"
```

Wrong: indentation inconsistent

Conditions in Python

- Booleans: `True` (with an uppercase **T**) is always verified and `False` is never verified
- Testing equality is done with `==` `!=` (which is variable assignment) 
- Combine conditions with `and` or with `or`
- Negate (take the opposite) with `not`

Operator	Description
<code>==</code>	Equality
<code>!=</code>	Inequality
<code>></code> , <code>>=</code> , <code><</code> , <code><=</code>	Other comparisons

Booleans



```
>>> mycondition = True
```

```
>>> print mycondition and False
```

A False B True C 0 D | E ERROR

```
>>> print not mycondition
```

A False B True C 0 D | E ERROR

```
>>> print not mycondition or False
```

A False B True C 0 D | E ERROR

```
>>> print not (mycondition and False
```

A False B True C 0 D | E ERROR

```
>>> print mycondition * 1
```

A False B True C 0 D | E ERROR

Conditions on Integers



```
>>> mynumber = 4
```

```
>>> print mynumber > 9
```

A False **B** True **C** 0 **D** | **E** ERROR

```
>>> print (mynumber = 10) or (mynumber = 11)
```

A False **B** True **C** 0 **D** | **E** ERROR

```
>>> print (mynumber == 10) or (mynumber == 11)
```

A False **B** True **C** 0 **D** | **E** ERROR

```
>>> print not (mynumber < 4)
```

A False **B** True **C** 0 **D** | **E** ERROR

```
>>> print mynumber < (mynumber + 4)
```

A False **B** True **C** 0 **D** | **E** ERROR

Iterating over Integers

- The `for` loop in Python can be used in many different ways
- One way to use it, is to iterate over integers
- Using `range(a, b)` which creates a range over all integers **starting with a and strictly smaller than b**
- The variable (below, I use `k`) used in the loop **will take the value of each integer** one after the other

```
for k in range(1, 10):  
    print k
```



```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Exercise Time



- Write an email with subject "CMPT 120, Class Exercise 1" or take a sheet of paper
- Write a Python program that prints the sum of squares from 1 to 100
 - initialize a variable to 0
 - iterate over all integers from 1 to 100 and add the square of that integer to your variable
 - print your variable
- Write a Python program that, assuming there is a variable called `myinteger`, that contains an integer (you don't know which one), prints "Even" if the number is even and "Odd" if it is odd
 - use the modulo/remainder operator `%`

Pacing and Understanding

How well did you understand today?



- A** Too easy, this lecture is way below my abilities
- B** Everything went at a good pace, and I am fine
- C** Too fast, but I will catch up on my own
- D** Too fast, and I need you to slow down
- E** I really do not think I can handle this