Extended Introduction to Computer Science
CS1001.py

Lecture 2: Python Programming Basics

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

Reminder: What we did in Lecture 1 (Highlights)

- Variables belong to classes that determine their types.
- We saw the types `'int'`, `'float'`, `'str'`.
- Different types allow different operations.
- Some operations allow "mixing" variables of different types.
- Subsequent assignments to the same variable can change its value and even its type.

Lecture 2: Planned Topics

1. More on variable types and operators (recap and extension of lecture 1)
   - Numeric and textual operators
   - Boolean variables and operators
   - Comparison operators
2. # Documenting your code with comments
3. Conditional Statements
4. Collections and iteration
   - `range`
   - Loops
   - Lists
   - Iterating over ranges, lists, and strings

1. More on variable types and operators

Variables and Assignments (reminder)

```
>>> e=2.718281828459045
>>> e
2.718281828459045
>>> pi=3.141592653589793
>>> pi
3.141592653589793
```

Variables with assigned values can be used as part of the evaluation of other expressions.

```
>>> e*pi+1
9.539734222673566
>>> e**pi+1
24.140692632779263  # ** stands for exponentiation
```

But assigning an expression with undefined variables leads to a run time error.

```
>>> e**(pi*i)+1
Traceback (most recent call last):
    File "<pyshell#4>", line 1, in <module>
    e**(pi*i)+1
NameError:  name 'i' is not defined
```

Python Programming Basics: Operations on Numbers

```
>>> 4
4
>>> type(4)
<class 'int'>
>>> 3.14159265358979323846264338327950288
3.141592653589793  # ouch! truncated...
>>> type(3.14159265358979323846264338327950288)
<class 'float'>  # floating point ("reals") type

>>> 8/5
1.6  #  / returns a float (real), the result of division
>>> 8//5
1  # // returns an integer, the floor of division
>>> 8%5 # % returns an integer, the remainder of division
3

>>> type(8/5)
<class 'float'>
>>> type(8//5)
<class 'int'>
```

Addition, subtraction, multiplication exist as well (mix, match, try!).

More Operations on Numbers and Casting

```
>>> 4+3.14
7.140000000000001
>>> 4/3.14
1.2738853503184713
>>> 4*3.14
12.56
>>> 3.14**4  # ** is exponentiation
97.21171216000002
```

In arithmetic operations mixing integers and floating point numbers, the result is typically a floating point number (changing the type this way is termed coersion or casting).

```
>>> 3.14*0
0.0
>>> 3.14/0
Traceback (most recent call last):
    File "<pyshell#8>", line 1, in <module>
    3.14/0
ZeroDivisionError:  float division by zero
```

Python Programming Basics: Strings

Variables could be of any type that Python recognizes. For example

```
>>> mssg1="Let there"
>>> mssg2=" be light"
>>> mssg1+mssg2
'Let there be light'
```

Variables of different types can sometimes be combined together, depending on the rules of the relevant types.

```
>>> n=2
>>> mssg1 + n*mssg2
'Let there be light be light'
```

But trying to add an integer to a string is not a good idea:

```
>>> mssg1+mssg2+n
Traceback (most recent call last)
    File "<pyshell #46>", line 1, in <module>
    mssg1+mssg2+n
NameError:  Can't convert 'int' object to str implicitly
```

Note that the error message gives you some information, albeit partial, on the source of the error, and is helpful in debugging it.

The Empty String

We saw a number of built-in strings methods, some of these methods' names have `str.` as their prefix. A not so intuitive, yet very important, is the empty string. It is invisible, yet it exists.

```
>>> ""
' '
>>> print("")
```

In some sense it is to strings like what the number 0 is to numbers.

```
>>> "Dr." + "Suess"
'Dr.Suess'
>>> "Dr." + "" + "Suess"  # "" is the empty string
'Dr.Suess'
>>> "Dr." + " " + "Suess"  # " " is a single space
'Dr. Suess'
```

The Empty String

We saw a number of built-in strings methods, some of these methods' names have `str.` as their prefix. A not so intuitive, yet very important, is the empty string. It is invisible, yet it exists.

```
>>> ""
' '
>>> print("")
```

In some sense it is to strings like what the number 0 is to numbers.

```
>>> "Dr." + "Suess"
'Dr.Suess'
>>> "Dr." + "" + "Suess"  # "" is the empty string
'Dr.Suess'
>>> "Dr." + " " + "Suess"  # " " is a single space
'Dr. Suess'

>>> "Bakbuk Bli Pkak "*4  # * denotes repetition
'Bakbuk Bli Pkak Bakbuk Bli Pkak Bakbuk Bli Pkak Bakbuk Bli Pkak '
>>> "Hello World"*0
''  # the empty string
>>> "Hello World"*-3
''  # the empty string, again
```

You'll meet the empty string again in the Computational Models course.

Python Basics: Boolean Variables

Boolean variables are either True or False. Note that Python's `True` and `False` are capitalized, while in most other programming languages they are not.

```
>>> a = True
>>> b = True
>>> c = False
>>> d = False
```

Boolean Variables and Boolean Operators

Boolean variables are either True or False. Note that Python's `True` and `False` are capitalized, while in most other programming languages they are not.

```
>>> a = True
>>> b = True
>>> c = False
>>> d = False
```

The standard Boolean, or logical, operators can be applied to them and generate complex Boolean expressions from "atomic" ones. Example of such operators are `and`, `or`, and `not`.

```
>>> a and b
True
>>> a and c
false
>>> a or c
True
>>> not a
false
```

More Boolean Variables and Operators

The operators and , or , and not are universal. This means that any Boolean operator (on one, two, or more variables) can be expressed using combinations of and , or , and not . (You will see this claim, plus possibly a proof of it, in the Computers Structure course.)

For example, Python does not have a built in Boolean xor (exclusive or) operator. Yet, with the following combination of the and , or , and not operators, we can express it:

```
>>> a = True
>>> b = True
>>> c = False
>>> d = False
>>> (a and (not b)) or ((not a) and b) # a xor b
False
>>> (c and (not d)) or ((not c) and d) # c xor d
False
>>> (a and (not c)) or ((not a) and c) # a xor c
True
```

Yet More Boolean Variables and Operators

```
>>> a = True
>>> b = True
>>> c = False
>>> d = False
```

It is not hard to see (*e.g.* by constructing truth tables) that the following combination of and , or , and not indeed produces the logical xor operator:

```
>>> (a and (not b)) or ((not a) and b)
False
>>> (c and (not d)) or ((not c) and d)
False
>>> (a and (not c)) or ((not a) and c)
True
```

Yet More Boolean Variables and Operators

```
>>> a = True
>>> b = True
>>> c = False
>>> d = False
```

It is not hard to see (*e.g.* by constructing truth tables) that the following combination of and , or , and not indeed produces the logical xor operator:

```
>>> (a and (not b)) or ((not a) and b)
False
>>> (c and (not d)) or ((not c) and d)
False
>>> (a and (not c)) or ((not a) and c)
True
```

It is annoying and time consuming to write and rewrite the same expression with different variables. We will address this issue when we discuss functions (in the next lecture).

Python Comparison Operators

Comparing numbers is important in many contexts. Python's comparison operators are intended for that: They operate on two numbers and return a Boolean value, indicating equality, inequality, or a size comparison.

```
>>> 5 == 5
True
>>> 6 != 6    # != is inequality, namely 6 ≠ 6 here
False
>>> 3 > 2
True
>>> 4 < 3
False
>>> 3 >= 2
True
>>> 4 <= 3
False
```

More Comparison Examples

We can compare numbers of different types. The interpreter implicitly coerces the more restricted type to the wider one, and performs the comparison.

```
>>> 19.0 > 7      # comparing an integer and a real number
True
>>> 14.0 == 14    # checking equality of an integer and a real number
???  # Don't be lazy - check it yourselves!
```

What Else Can We Compare?

We can compare integers and floating point numbers.
What about comparing Booleans?

Well, instead of guessing, let us try.

What Else Can We Compare?

We can compare integers and floating point numbers.
What about comparing Booleans?

Well, instead of guessing, let us try.

```
>>> True > False
True
>>> True > True
False
>>> 1 > True
False
>>> False > 0
False
>>> False > -1
True
>>> False > 2.17
False
```

Given these examples, how do you think the interpreter "views" True and False?

More on True and False

More on True and False

```
>>> 4+True
5
>>> True*2
2
>>> True+False
1
>>> True==1
True
>>> True==0
False
>>>
```

So the Python interpreter "views" True as 1 and False as 0.

Yet, we strongly recommend you do not use True and False in arithmetic contexts!

What Else Can We Compare?

What about comparing strings?

What Else Can We Compare?

What about comparing strings?

```
>>> "Benny" > "Amir"
True
>>> "Amir">"Benny"
False
>>> "Amir">"amir"
False
>>> "0">"1"
False
>>> "0">"?"
False
>>> "0">"" # the empty string
True
>>> ">" "   # a string containing exactly one space
False
```

So how do you think the interpreter compares strings?

What Else Can We Compare?

What about comparing strings?

```
>>> "Benny" > "Amir"
True
>>> "Amir">"Benny"
False
>>> "Amir">"amir"
False
>>> "0">"1"
False
>>> "0">"?"
False
>>> "0">"" # the empty string
True
>>> ">" "   # a string containing exactly one space
False
```

So how do you think the interpreter compares strings?

```
>>> "0"> 0
Traceback (most recent call last):
    File "<pyshell#19>", line 1, in <module>
    "0">0
TypeError:  unorderable types:  str() > int()
```

Documenting Your Programs

An essential part of writing computer code is documenting it.
For you to later understand what you had in mind when typing some commands.
For your teammate at work to be able to coordinate her or his code with yours.
For the grader or teaching assistant in your class to try and understand your code, and grade it accordingly.
Etc. etc.

Documenting Your Programs with Comments

To assist in documentation, all programming languages have comments. Comments are pieces of text that are not interpreted or executed by the computer. The simplest type of comments in Python are one line comments. They start with the hash character, #, and extend to the end of the physical line. Note that # within a string is treated as a character and not as a beginning of a comment.

```
# comments can start at beginning of a line
a=1 # comments can also start after the beginning of a line
"# but this is a string, NOT a comment"

>>>"# but this is a string, NOT a comment"
'# but this is a string, NOT a comment'
```

2. Conditional statements

Python Programming Basics: Conditional Statements

The flow of very simple programs is linear: We execute one statement after the next, in the order they were typed. However, the flow of most programs depends on values of variables and relations among them. Conditional statements are used to direct this flow.

```
>>> if 2001 % 71 == 0:    # 2001 % 71 is remainder of 2001 divided by 71
        print("2001 is divisible by 71")
    else:
        print("2001 is not divisible by 71")

2001 is not divisible by 71
```

Syntax of Conditional Statements

```
>>> if 2001 % 71 == 0:    # 2001 % 71 is remainder of 2001 divided by 71
        print("2001 is divisible by 71")
    else:
        print("2001 is not divisible by 71")

2001 is not divisible by 71
```

Important syntactic points: The colon following the `if` statement acts as to open a new scope (opening a parenthesis, or `begin`, in other programming languages).
The `print` statement in the line below it is indented one tab to the right. This indicates it is within the scope of this `if` .
The first statement not indented this way (in our case, the `else`) is outside that scope.

In IDLE, you can use tabs or spaces for indentation. The width of the tab / number of spaces must be consistent whithin a single scope. Consistency across the whole program is `highly recommended` as well. However sometimes inconsistency is practically unavoidable (can you think of such a scenario?).

Formal Definition: Conditional Statements

If in doubt, ask for help:

```
>>> help("if")
The ‘‘if‘‘ statement
*******************

The ‘‘if‘‘ statement is used for conditional execution:

        if_stmt ::= "if" expression ":" suite
                    ( "elif" expression ":" suite )*
                    ["else" ":" suite]

It selects exactly one of the suites by evaluating the expressions one
by one until one is found to be true (see section *Boolean operations*
for the definition of true and false); then that suite is executed
(and no other part of the ‘‘if‘‘ statement is executed or evaluated).
If all expressions are false, the suite of the ‘‘else‘‘ clause, if
present, is executed.

Related help topics:  TRUTHVALUE
```

Nested Conditional Statements

Conditional statements can be nested. And the scope of a conditional statement can have more than a single statement within it.

```
>>> n=2**97 + 100

>>> if n % 4 != 0:
        print("n=",n)
        print("n is not divisible by 4")
    else:
        if n % 8 == 0:      # nested "if"
            print("n=",n)
            print("n is divisible by 8")
        else:               # nested "else"
            print("n=",n)
            print("n is divisible by 4 but not by 8")

n= 158456325028528675187087900772
n is divisible by 4 but not by 8
```

3. Collections and iteration

Iteration: Definition

Iteration means the act of repeating a process usually with the aim of approaching a desired goal or target or result. Each repetition of the process is also called an "iteration," and the results of one iteration are used as the starting point for the next iteration.

(taken from Wikipedia.)

Iteration

It is very common to have a portion of a program where we wish to iterate the same operation, possibly with different arguments. For example,

```
>>> 1+2+3+4+5+6+7+8+9+10
55
```

If we just want to add the numbers from 1 to 10, the piece of code above may be adequate. But suppose that instead, we wish to increase the summation limit to 100, $1,000$, or even $1,000,000$.

We obviously ought to have a more efficient method to express such iterated addition. We want to iterate over the integers in the range [1,100], [1,1000], or [1,1000000], repeatedly adding the next element to the partially computed sum.

Python's Range

Python has a built in construct, `range`, which creates an ordered collection of all integers in a given range. For two integers a, b, `range(a,b)` contains all integers k satisfying $a \leq k < b$. Note that if $a \geq b$, `range(a,b)` is empty (however, this is not an error).

Special case: `range(b)` is a shorthand for `range(0,b)`.

```
>>> range(1,90)
range(1, 90)
>>> type(range(1,90))
<class 'range'>
```

`range` returns an iterator that generates the numbers in the range on demand (text from `help(range)`).

More Python's Range

More generally, `range` can create arithmetic progressions.
For three integers a, b, d with $d > 0$, `range(a,b,d)` contains all integers of the form $a + id,\ i \geq 0$, satisfying $a \leq a + id < b$.
So we can see that `range(a,b)` is a shorthand for `range(a,b,1)`.

For $d < 0$, `range(a,b,d)` contains all integers of the form $a + id,\ i \geq 0$, satisfying $b < a + id \leq a$.

Note that for $d = 0$, `range(a,b,d)` results in an error.

Range and Iteration: `for`

Suppose we wish to compute $1 + 2 + \ldots + 100$. The appropriate range containing these numbers is `range(1,101)`.

```
>>> partial_sum = 0
>>> for i in range(1,101):
        partial_sum = partial_sum+i

>>> partial_sum
5050
```

More Range and Iteration: `for`

```
>>> partial_sum = 0
>>> for i in range(1,101):
        partial_sum = partial_sum+i
>>> partial_sum
5050
```

In this statement, `for` and `in` are reserved words of Python, causing iterated execution of the addition, where the variable i goes over all elements in `range(1,101)`.
Notice the colon and indentation, which determine the scope where the iteration occurs.

Range and Iteration: `sum`

Summation over a structure is so common that Python has a built-in function for it, `sum`, enabling an even more concise code. For example, the following code computes $1 + 2 + \ldots + 100$:

```
>>> sum(range(1,101))
5050
```

Historic, Non-Iterative Digression

Do you know young Carl Friedrich (Gauss, April 1777 – February 1855)? At the age of six, he allegedly figured out how to efficiently compute the sum $1 + 2 + \ldots + 100$.

(painted by Christian Albrecht Jensen – taken from Wikipedia.)

A Computational Digression

(painted by Christian Albrecht Jensen – taken from Wikipedia.)

Gauss' observation was that $1 + 2 + \ldots + n = n(n+1)/2$. We are not going to use it, though, and simply let the computer chew its way through the iteration.

Young Gauss' observation enabled him to calculate the sum $1 + 2 + \ldots + 100$ very fast. In modern terminology, we could say that his method, requiring only one addition, one multiplication and one division, is much more efficient computationally than our method, requiring exactly $n - 1$ addition operations.

We will define these notions in a precise manner in one of the next lectures.

Remarks on Gauss vs. `sum`

```
>>> sum(range(100000000))
4999999950000000
```

Computing this sum takes $100000000 = 10^8$ operations. Without exactly measuring it, the time taken for the computation to complete is noticeable.

```
>>> (100000000-1)*100000000//2
4999999950000000
```

Computing this sum "the Gauss way" requires 3 arithmetic operations, much fewer than 10^8, and is noticeably faster.

Good algorithms often succeed to tackle problems in non-obvious ways, and dramatically improve their efficiency.

Back to Iteration: `while`

The following code finds and prints all divisors of $n = 100$.

```
>>> n = 100
>>> m = 100
>>> while m>0:
        if n%m == 0:
            print(m)
        m = m-1

100
50
25
20
10
5
4
2
1
```

The loop is entered and executed as long as the condition, `m>0`, holds.

Note that if you fail to include the `m=m-1` assignment, the execution will enter an infinite loop.

Aborting Iteration: break

```
>>> for i in range(1,10):
        print(i)
        if i % 7 == 0:     i % 7 is remainder of i divided by 7
            break

1
2
3
4
5
6
7
```

break terminates the nearest enclosing loop, skipping any other code that follows the break inside the loop. It may only occur syntactically nested in a for or a while loop. Useful for getting out of loops when some predefined condition occurs.

Basic Data Structures: Lists

A list is an ordered sequence of elements. The simplest way to create a list in Python is to enclose its elements in square brackets.

```
>>> my_list = [2,3,5,7,11]
>>> my_list
[2, 3, 5, 7, 11]
```

Lists are Indexable

List elements have indices, enabling direct (aka "random") access. In this respect, lists are similar to arrays in other prog. languages (yet they differ in other aspects, causing them in general to be less efficient, but this is noticeable only in "heavy" computational tasks).

Indices in lists start with 0, not with 1.

```
>>> my_list=[2,3,5,7,11]
>>> my_list[0]
2
>>> my_list[4]
11
>>> my_list[5]
Traceback (most recent call last):
    File "<pyshell#7>", line 1, in <module>
        my_list[5]
IndexError:  list index out of range
```

Length of Lists

`len` returns the length (number of elements) of a list.

```
>>> len(my_list)
5
>>> len([])
0
```

Quite often, the length is a useful quantity to have.

List Comprehension

In mathematics, concise notations are often used to express various constructs, such as sequences or sets. For example, $\{n \mid n \in \mathbb{N} \text{ is prime and } n < 60\}$ is the set of prime numbers that are smaller than 60. Incidentally, this set is $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59\}$.

List Comprehension

In mathematics, concise notations are often used to express various constructs, such as sequences or sets. For example, $\{n \mid n \in \mathbb{N} \text{ is prime and } n < 60\}$ is the set of prime numbers that are smaller than 60. Incidentally, this set is $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59\}$.

Python supports a mechanism called list comprehension, which enable syntactically concise ways to create lists. For example

```
>>> list1=[i**2 for i in range(1,11)]
>>> list1
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

With a test for primality (which we'll soon develop), we can use list comprehension to generate the list of all primes smaller than 60:

```
>>> primes60=[n for n in range(2,60) if is_prime(n)]
>>> primes60
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

Iterating over Lists

Lists are ordered sequences. As such, we can iterate over their elements much in the same way we iterate over a `range`.

```
>>> list1 = [i**2 for i in range(1,11)]
>>> list1
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> partial_product = 1
>>> for k in list1:
        partial_product = partial_product * k

>>> partial_product
13168189440000
```

And now, a tiny variant (find the differences):

```
>>> list2=[i**2 for i in range(0,11)]
```

```
>>> partial_product2=1
>>> for k in list2:
        partial_product2 = partial_product2 * k

>>> partial_product2
0                    # this should come as no surprise
```

More Iterations over Lists

We can iterate over the elements of any list, not only lists of numbers. For example, the following list consists of strings:

```
>>> list3 = ["Rattus norvegicus, ", "Alligator mississippiensis, ","Mus
musculus, ", "Bos taurus, ", "Ornithorhynchus anatinus, ", "Pan
troglodytes "]
>>> question = "Who is exceptional:  "
>>> for species in list3:
        question = question + species  # + is concatenation in strings
>>> question = question + "?"
>>> print(question)                    # what will the outcome be?
```

More Iterations over Lists

We can iterate over the elements of any list, not only lists of numbers. For example, the following list consists of strings:

```
>>> list3 = ["Rattus norvegicus, ", "Alligator mississippiensis, ","Mus
musculus, ", "Bos taurus, ", "Ornithorhynchus anatinus, ", "Pan
troglodytes "]
>>> question = "Who is exceptional:  "
>>> for species in list3:
        question = question + species  # + is concatenation in strings
>>> question = question + "?"
>>> print(question)                    # what will the outcome be?
```

```
Who is exceptional:  Rattus norvegicus, Alligator mississippiensis, Mus
musculus, Bos taurus, Ornithorhynchus anatinus, Pan troglodytes ?
```

Iterating over Strings

Strings, too, are ordered sequences of characters. As such, we can iterate over their elements much in the same way we iterate over a `list`.

```
>>> question= "Who is exceptional:  Rattus norvegicus, Alligator
mississippiensis, Mus musculus, Bos taurus, Ornithorhynchus anatinus, Pan
troglodytes ?"
>>> stutter=""
>>> for letter in question:
        stutter=stutter+letter*2
```

Iterating over Strings

Strings, too, are ordered sequences of characters. As such, we can iterate over their elements much in the same way we iterate over a `list`.

```
>>> question= "Who is exceptional:  Rattus norvegicus, Alligator
mississippiensis, Mus musculus, Bos taurus, Ornithorhynchus anatinus, Pan
troglodytes ?"
>>> stutter=""
>>> for letter in question:
        stutter=stutter+letter*2

>>> print(stutter)
WWhhoo iiss eexxcceeppttiioonnaall::  RRaattttuuss nnoorrvveeggiiccuuss,,
AAlllliiggaattoorr mmiissssiissssiippppiieennssiiss,, MMuuss
mmuussccuulluuss,, BBooss ttaauurruuss,, OOrrnniitthhoorrhhyynncchhuuss
aannaattiinnuuss,, PPaann ttrrooggllooddyytteess ??
```

We will see many other data structures in this course.