DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, September 14, 2020

Python basics

Python is an **interpreted** language.

Programs are loaded and run rather than being compiled first.

We will use the IDLE editor that is available with the Python download on the Python site: http://www.python.org

To see how to interpret a Python program, consider the Hello World program:

```
>>> print('Hello world!')
Hello world!
```

There are links on the D2L site that give information about Python. Please make sure that you have reviewed them, and install Python as soon as possible if you will be working at home on programs.

Assignments, basic types, and identifiers

Python is a **dynamically typed language**.

This means that we do not have to declare the type for a variable prior to using it. In fact, you create a variable by assigning it.

```
>>> x
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> x = 3
>>> x
3
```

Note the initial error message above. The variable name x is undefined until it is assigned. Any attempt to use the variable before it has been defined (by using it in an assignment) will result in an exception.

Note that **the same variable can be used for different types** without any issues. The type for the variable is taken from the value stored in it:

```
>>> x = 3
>>> x
3
>>> type(x)
<class 'int'>
>>> x = 2.7
>>> x
2.7
>>> type(x)
<class 'float'>
>>> x = 'hello'
>>> x
'hello'
>>> type(x)
<class 'str'>
```

Python allows you to assign two variables a and b using one statement:

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
```

A value can also be assigned to several variables simultaneously:

```
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
```

Basic Python types

What are the **basic types in Python**?

The basic Python built-in types include:

- Numeric
  - Integer
  - Boolean
  - Floating point
- String

Integers are whole numbers and **floating point numbers** are decimals.

They can be combined using the **arithmetic operators**: +, -, *, /, //, and % and using parentheses:

Consider some examples:

```
>>> 3+7
10
>>> 12/5
2.4
>>> 12 // 5
2
```

```
>>> 12 % 5
2
>>> 3*2+1
7
>>> (3-1)*(4+1)
10
>>> 4.321/3+10
11.440333333333333
>>> 4.321/(3+10)
0.3323846153846154
>>> 2**3
8
```

Boolean expressions evaluate to either true or false rather than a numerical value.

Consider the following examples:

```
>>> 3 < 2
False
>>> 3 > 2
True
>>> x = 4
>>> x == 4
True
>>> x == 3
False
>>> x > 4
False
>>> x <= 4
True
>>> x != 5
True
```

Testing whether two expressions have the same value is done using the double equality (==) operator. To check whether two expressions are not equal, the not equals operator (!=) is used.

Boolean expressions can be combined together using the **Boolean operators** and, or, and not.

The and operator applied to two Boolean expressions will evaluate to True if both expressions evaluate to True. If either expression evaluates to False, then it will evaluate to False.

```
>>> 2 < 3 and 4 > 5
False
>>> 2 < 3 and True
True
```

The or operator applied to two Boolean expressions evaluates to False only when both expressions are false. If either one or both are True, then it evaluates to True.

```
>>> 2 < 3 or 4 > 5
True
```

```
>>> 3 < 2 or 2 < 1
False
```

The not operator is a unary Boolean operator, which means that it is applied to a single Boolean expression. It evaluates to False if the expression it is applied to is True, or to True if the expression is False.

```
>>> not (3 < 4)
False
```

See page 20 for the truth tables for the Boolean operators.

Strings are used to represent character data.

String values can be represented as a **sequence of characters**, including blanks or punctuation characters, **enclosed within single quotes**.

Python provides a number of operations that can be performed on string values.

The + operator, when used on two strings, returns a new string that is the concatenation of the two strings.

The * operator can be applied to a string and an integer to produce as many concatenations of the string as the number:

```
>>> 'one' + 'two'
'onetwo'
>>> a + b
'helloworld'
>>> a + ' ' + b
'hello world'
>>> 'hello' * 'world'
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in
<module>
    'hello' * 'world'
TypeError: can't multiply sequence by
non-int of type 'str'
>>> 'hello' * 3
'hellohellohello'
>>> 30 * '-'
'------------------------------'
```

The individual characters of a string can be obtained using the **indexing operator []**.

The indexing operator takes a **non-negative index** i and returns the character of the string at offset i:

```
>>> a = 'hello'
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[2]
'l'
```

Substrings of a string can be obtained using the indexing operator:

```
>>> name = 'Amber'
>>> name[0]
'A'
>>> name[0:2]
'Am'
>>> name[3:4]
'e'
>>> name[1:4]
'mbe'
```

Negative indices can be used to access the characters from the back of the string. For example, the last character can be obtained as follows:

```
>>> name[-1]
'r'
```

Casting between types

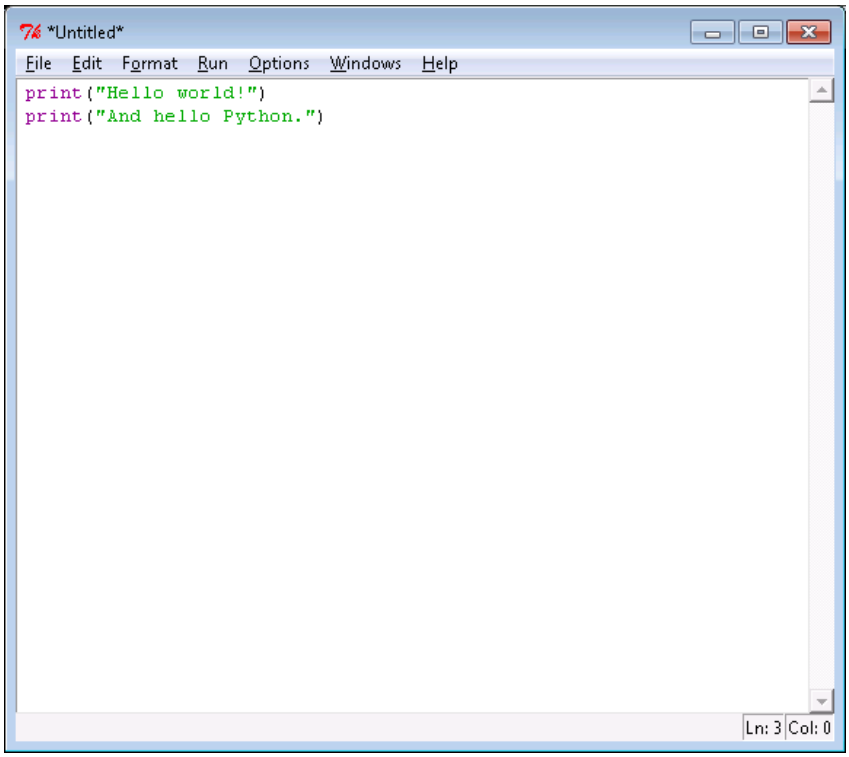
You can explicitly cast between compatible types:

```
>>> x = 3
>>> x = 3.1415
>>> y = int(x)
>>> y
3
>>> z = True
>>> y = int(z)
>>> y
1
>>> x = '1234'
>>> y = int(x)
>>> y
1234
>>> x = float(y)
>>> x
1234.0
```

Python files

A **Python program is a text file** containing Python statements. We will use the built-in editor attached to IDLE for this course. While it is basic, it will provide all the functionality we need.

To open the IDLE editor, click on 'File ' and then 'New Window '. Then type in the program you wish to save:



To execute the program, click F5 (or 'Run ' and then 'Run Module ').

> You will be asked to save the program in a file. The file name must have the suffix '.py '.

IDLE is a Python programmer's editor, with features that are helpful for Python programming, including: automatic indentation, abilities to run/debug Python code from within the editor, etc.

Interactive input

In order to solve a wider variety of problems, executing programs often need to interact with the user.

> The input function is used by the program to request input data from the user. Whatever the user types will be treated as a string.

Consider the following program found in the file **input.py**:

```
x = input("Enter x: ")
print("x is", x)
```

When this program is run it produces the following sample output:

```
>>>
```

```
Enter x: Hello!!
x is Hello!!
>>> ================================ RESTART
================================
>>>
Enter x: 17
x is 17
>>> x
'17'
>>> x+4
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    x+4
TypeError: Can't convert 'int' object to str
implicitly
```

Note that after we entered 17 and tried to add 4 to x, we got an error. It is because the 17 we typed is treated as a string, not a number.

In order to force Python to evaluate a string, you need to use the eval function:

```
>>> y = eval('17')
>>> y+4
21
```

So we can modify the program above to produce a new one that reads in expressions that should be evaluated found in the file **input2.py**:

```
x = eval (input ("Enter x: "))
print("x is", x)
```

When this program is run it produces the following sample output:

```
>>>
Enter x: 17
x is  17
>>> x+4
21
```

Functions

Throughout the quarter we will take a procedural approach to developing Python programs.

This means that we will use functions to encapsulate important tasks and then use those functions in other more complex problems.

We have already seen some built-in functions.

For example:

- **len**() takes a string or list as input and returns the length of the string or list
- **print**() takes a string prints it to the screen

Functions can take more than one parameter. For example, **min**() is an example of a function that takes multiple inputs and returns the smallest.

```
>>> min(2,5)
2
>>> min(2, -1, 5)
-1
```

The function f(x) = x + 1 can be defined as a Python function as follows:

```
def f(x):
      return x+1
```

In general, **a Python def statement** has the following format:

```
def <function name> (<0 or more parameters>):
      <iterated statements>
      # one or more Python statements,
      # all with a tab in front of them
```

The **def keyword** is used to define a function.
Following the def command is the **name of the function**. (In the case above the name of the function is f).
Following the name is a **set of parentheses containing 0, 1, or more input arguments**. (In the case above we have one argument x).
A **colon** ends the first line.

Below the def statement and indented with a tab is **the implementation of the function**, i.e. the Python code that does the work.

The **optional return statement** is used to specify the value obtained when the function is evaluated in an expression. (In the case above, the value x+1 is returned).

In order to run a function on some input, we use it like we would any other Python function:

```
>>> f(3)
4
>>> 3*f(2) + 1
10
```

Help on functions

The built-in functions we have seen so far all have documentation that can be viewed with the **help function**.

The following is an example of how to use the help function:

```
>>> help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or
mapping.

>>> help(min)
Help on built-in function min in module builtins:

min(...)
    min(iterable[, key=func]) -> value
    min(a, b, c, ...[, key=func]) -> value

    With a single iterable argument, return its
smallest item.
    With two or more arguments, return the smallest
argument.
```

Unfortunately, the help function does not provide useful information about programmer-defined functions unless we have provided information about the function.

For example, we can see what it displays when we ask about the function f we defined in the previous section:

```
>>> help(f)
Help on function f in module __main__:

f(x)
```

In order to get a documented function, we need to add a string (called a **doc string**) below the def statement:

```
def f(x):
      'f prints x+1'
      print(x+1)
```

Then when we call the help function on the method, we will get the string we have added:

```
>>> help(f)
Help on function f in module __main__:

f(x)
    f prints x+1
```

Common mistake

A common mistake when writing functions is to use print instead of return.

For example, if we define the function f above as:

```
def f(x):
      print(x+1)
```

then the function f will print x+1 on input x, but when used in an expression, it will not evaluate to it.

See the sample runs below to see what is meant by this:

```
>>> def f(x):
      print(x+1)


>>> f(3)
4
>>> 3*f(2)+1
3
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    3*f(2)+1
TypeError: unsupported operand type(s) for *: 'int'
and 'NoneType'
```

When evaluating f(3), the Python interpreter evaluates f(3) which will print 4 but does not evaluate to anything.

The second example makes this clear. The Python interpreter will print 3 when evaluating f(2), but f(2) does not evaluate to anything and so cannot be used in an expression.

Exercise: Does the `print()` function return anything?

Answer: The best way to check is to use an assignment statement so that the value returned by print gets a name:

```
>>> x = print("Hello!")
Hello!
>>> x
>>>
```

We can see above that x does not have a value, meaning that the print() function does not return anything.

Lists

Lists are ordered collections of objects.

Lists have the following **properties**:

- The objects are ordered left to right
- Each object in the list is accessible with an index, with the leftmost object having index 0 and every other object having index one more than the index of the object to its immediate left
- Like strings, list support indexing, slicing, and concatenation operations

- Unlike strings, lists can change, i.e. they are **mutable**
- Lists have dynamic lengths, meaning that they can grow and shrink as objects are added or deleted from the list

The following demonstrates some simple list operations:

```
>>> l = []
>>> l1 = [1, 2, 3, 5, 7, 11, 13]
>>> l2 = [l1, 2, 4, 6, ["hello", "goodbye"]]
>>> l2
[[1, 2, 3, 5, 7, 11, 13], 2, 4, 6, ['hello',
'goodbye']]
>>> len(l)
0
>>> len(l1)
7
>>> len(l2)
5
>>> l1 + l1
[1, 2, 3, 5, 7, 11, 13, 1, 2, 3, 5, 7, 11, 13]
>>> ["hello"] * 3
['hello', 'hello', 'hello']
>>> l1 * 2
[1, 2, 3, 5, 7, 11, 13, 1, 2, 3, 5, 7, 11, 13]
>>> 2 in l1
True
>>> 4 in l1
False
```

The following demonstrates indexing and slicing:

```
>>> l1[2]
3
>>> l1[2:5]
[3, 5, 7]
>>> l1[-1]
13
```

These examples demonstrate how to change lists:

```
>>> l1
[1, 2, 3, 5, 7, 11, 13]
>>> l1[5] = 17
>>> l1
[1, 2, 3, 5, 7, 17, 13]
>>> l1.append(17)
>>> l1[5] = 11
>>> l1
[1, 2, 3, 5, 7, 11, 13, 17]
>>> l1.reverse()
>>> l1
[17, 13, 11, 7, 5, 3, 2, 1]
>>> l1.sort()
>>> l1
[1, 2, 3, 5, 7, 11, 13, 17]
>>> del l1[3]
>>> l1
[1, 2, 3, 7, 11, 13, 17]
>>> del l1[2:4]
>>> l1
[1, 2, 11, 13, 17]
```

Exercise: Consider the following list of student homework grades

```
>>> grades = [9, 7, 7, 10, 3, 9, 6, 6, 2]
```

Write Python expressions to:

- Find the number of 7 grades
- Change the last grade to 4
- Find the maximum grade
- Sort the grades list
- Find the average grade

Hint: See Table 2.2 on page 28 of the book for some list operations and Table 2.3 on page 30 for some list functions.

The solution is in **changeLists.py**.

Precedence

An expression is a combination of numbers (or other objects) and operators that is evaluated by the Python interpreter to some value.

As a review, below is a list of operators given in order of operator from lowest precedence to highest precedence:

Operator	Description
lambda	Lambda expression
or	Boolean OR
and	Boolean AND
not *x*	Boolean NOT
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, including membership tests and identity tests,
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, //, %	Multiplication, division, remainder
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation [8]
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference

Operator	Description
(expressions...), [expressions...], {key:datum...}, `expressions...`	Binding or tuple display, list display, dictionary display, string conversion

Using the rules above, describe what value the follow expressions will produce:

- 2 + 3 * 5
- 32 // 3 ** 2
- 3 / 5 * 8

Iteration structures

A fundamental programming tool is the ability to repeat a block of code.

For example, the program may need to iterate through a list or other collection of objects and execute one or more statements on the objects in the list.

As an example, consider the following:

```
>>> L = ['cat', 'dog', 'chicken']
>>> for animal in L:
      print (animal)


cat
dog
chicken
```

Consider another example:

```
>>> L = [67, 34, 98, 63, 23]
>>> for num in L:
    print(num)

67
34
98
63
23
```

And consider yet another example:

```
>>> s = "Puccini"
>>> for c in s:
      print(c)

P
u
c
c
i
n
i
```

In general, a for loop has the following structure:

```
for <var> in <sequence>:
      <body>
```

<sequence> is an object whose type is one of the sequence types (e.g. string, list, etc.). It contains an ordered collection of items.

<var> is a valid identifier.

When Python runs a for loop, it will assign successive values in <sequence> to <var> and execute the statements in <body> for each value.

The range function

Sometimes we want to iterate over a sequence of numbers in a certain range, rather than an explicit list.

For example, we may want to repeat **print(i) for all values of i**:

a. From 0 to 9, i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
b. From 0 to 1, i.e. 0, 1
c. From 3 to 12, i.e. 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
d. From 0 to 9, but with a step of 2, i.e. 0, 2, 4, 6, 8
e. From 0 to 24 with a step of 3, i.e. 0, 3, 6, 9, 12, 15, 18, 21, 24
f. From 3 to 12 with a step of 5, i.e. 3, 8

Let's see how to solve some of these examples:

```
>>> for i in range(10):
      print(i, end=" ")


0 1 2 3 4 5 6 7 8 9
>>> for i in range(2):
      print(i, end=" ")


0 1
>>> for i in range(3,13):
      print(i, end=" ")


3 4 5 6 7 8 9 10 11 12


>>> for i in range(0,10,2):
      print(i, end=" ")


0 2 4 6 8
```

```
>>> for i in range(0,24,3):
      print(i, end=" ")


0 3 6 9 12 15 18 21
>>> for i in range(3, 12, 5):
      print(i, end=" ")


3 8
```

Exercises

Let's do some exercises to solidify our understanding of functions, input, and lists.

Exercise 1: To practice functions and lists, consider the following exercise: Write a function that takes as a parameter a list and prints the first and the last elements in the list to the screen.

For example:

```
>>> printList([1, 2, 3, 4, 5])
1
5
>>> printList(['one', 'two', 'three'])
one
three
>>>
```

What if the function gets an empty list?

To fix it we need to use an if statement. Demonstrate how to do that.

See the solutions in **functions.py**.

Exercise 2: Write for loops that will print the following sequences:

a. 0
b. 0 1 2 3 4
c. 1 2 3 4
d. 3 4 5 6
e. 1
f. 0 3
g. 0 2 4 6 8 10 12 14 16 18
h. 17 13 9 5

See the solution in **forloops.py**.

Exercise 3a: Write a program that requests a positive integer n from the user and outputs all even positive integers up to and including n.

For example:

```
>>> evenNums()
Enter a whole number: 10
2 4 6 8 10
>>> evenNums()
Enter a whole number: 9
2 4 6 8
>>>
```

Note that you will need to do this without decision structures, since we haven't yet talked about conditionals. It can be done (easily)!

See the solution in **functions.py**.

Exercise 3b: Write a program that requests a positive integer n from the user and outputs all odd positive integers up to and including n.

For example:

```
>>> oddNums()
Enter a whole number: 10
1 3 5 7 9
>>> oddNums()
Enter a whole number: 9
1 3 5 7 9
>>>
```

See the solution in **functions.py**.

Exercise 4: Write a function that prints either the even or the odd numbers based on the string provided.

For example:

```
>>> printNums('even')
Enter a whole number: 10
2 4 6 8 10
>>> printNums('odd')
Enter a whole number: 10
1 3 5 7 9
>>>
```

For full credit, do not duplicate the work done in the previous exercises. Just call the right function!

Decision structures

The if statement is one of the basic ways that the Python interpreter can select which action to perform, based on a test.

The following tests whether the input argument is less than 0. If it is, it prints a message that the argument is negative. Otherwise, the function terminates.

```
if n < 0:
      print('n is negative ')
```

The **if statement** in its simplest form, has the following structure:

```
if <condition>:
      <body>
```

`<condition>` is a Boolean expression, i.e. an expression that evaluates to True or False.

Some examples of such conditions include:

- n < 0
- t >= 86
- name == 'Settle '
- c in x

`<body>` consists of one or more statements that are executed if the condition evaluates to True. These statements must be indented with respect to the if statement.

```
if <condition>:
      body statement 1
      body statement 2
      …
      last body statement
rest of the program here
```

Draw a **flow chart** for the if statement.

Consider the following **example program** (in **tests.py**):

```
if n < 0:
    print("n is negative")
    print("These indented statements are part")
    print("of the if statement body")
    print("Still in the body...")
  print("In the body no more")
  print("I am executed whether or not the condition")
  print("evaluates to True.")
```

Try sample runs on:

- Positive values (e.g. 4)
- Negative values (e.g. -4)

Three-way (and more) decisions

The most general form of an if statement allows for multi-way decisions.

For example, consider the following (in the file **tests.py**):

```
def test3(n):
    if n < 0:
        print("n is negative")
    elif n > 0:
        print("n is positive")
    else: print("n is 0")
```

This code has three alternative executions depending on whether n is positive, negative, or 0, seen below:

```
>>> test3(-4)
n is negative
>>> test3(0)
n is 0
>>> test3(4)
n is positive
```

The **general form of an if statement** is:

```
if <condition1>:
      <body1>
elif <condition2>:
      <body2>
elif <condition3>:
      <body3>
…
else:
      <last body>
```

Each condition has an associated body that is the block of code to be executed if the condition evaluates to True.

Each condition has an associated body that is the block of code to be executed if the condition evaluates to True.

Note that conditions do not have to be mutually exclusive, in which case the order in which they appear will make a difference.

Consider **temp2** in **tests.py**:

```
def temp2(t):
    if t >= 86:
        print("It is hot")
    elif t > 0:
        print("It is cool")
    else:
        print("It is freezing")
```

The above program is correct because of the order in which conditions are evaluated in Python.

Why would the version of the program below be incorrect? What value would produce unexpected results?

```
def temp3(t):
    if t > 0:
        print("It is cool")
    elif t >= 86:
        print("It is hot")
    else:
        print("It is freezing")
```