Writing simple programs

Read: *Chapter 2 from textbook*

- Start the command-line interpreter.
- Start again *IDLE* and open the text editor window. Open your file `chaos.py`.

We are going to study the various components of `chaos.py` more carefully.

- **Names (identifiers):** We have already used names for our modules, functions, variables. Some names cannot be used (*keywords*); look at Table 2.1 in the text. There are also specific rules for forming names (e.g., a name cannot contain space(s), it must start with a letter or '_' etc.); read Section 2.3.1.

  Variables are used to store values, as we have seen. In `chaos.py` there are 2 variables, x and i.
- **Expressions:** An expression is a piece of code that computes a new value (data). The simplest expressions are variables and literals[1]. Several expressions can be combined using operands to form more complex expressions.

  In `chaos.py` the piece of code 3.9*x*(1-x) is an expression; 3.9 is a literal (also an expression); x is a variable (also an expression). (1-x) is also an expression that is built by the literal 1 and the variable x combined by the arithmetic operator -; the parentheses are used in order to set the ordering in which the (sub-)expressions should be calculated (here (1-x) is calculated first and then 3.9*x*(1-x)). Type the following. The first line is an assignment statement (more about this later), but each one of the rest is an expression; each time, the interpreter will try to *evaluate* the expression to its value.

  ```
  >>> x = 5
  >>> x
  >>> 3.9*x*(1-x)
  >>> 32
  >>> “32”
  ```

  Notice that 32 evaluates to the *integer number* 32, but “32” evaluates to the *string* '32'. In the first case, your data is of **type** `int` (integer), and in the second case of type `str` (string). In general, you can mix only data of the same type in your expressions. For example, try the following:

  ```
  >>> 32+”32”
  ```

  You should get an error like the following:

  ```
  Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
  ```

1 A literal is a value itself, e.g., 5, 10.3, John (as opposed to, e.g., variables that contain a value).

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The last line tells you that operator + cannot be used to combine an `int`(eger) with a `str`(ing). It's like trying to combine apples and oranges! By the way, recall that you can use either single or double (but not both at the same time) quotes to define a string literal in Python. So, typing "32" or '32' makes no difference; they both evaluate to the sequence of characters '32' (a character (not number!) '3' followed by a character '2'). Now, if you try

```
>>> "32"+"32"
```

it should work fine. What do you get? Operator + apparently works differently when it is applied to strings than when it is applied to integers:

```
>>> 32+32
```

That's one major reason for knowing exactly the **type** of our values/expressions!

- **Assignments:** In the previous example, we said that

  ```
  >>> x = 5
  ```

  is an **assignment** statement. In general, an assignment statement has the following format:

  <variable> = <expression>

  What happens here is that first the <expression> in the RHS is evaluated, and then this value becomes the value stored in <variable>. Try the following:

  ```
  >>> x = 5
  >>> x
  >>> x = x+1
  >>> x
  ```

  Notice that variable x appears in both RHS and LHS in the third line. But it is not treated the same in both sides: In the RHS it participates in an expression so its *value* is used (since the expression is being evaluated); currently its value is 5, so the RHS evaluates to 6, and THEN this new value (6) becomes the value of the variable in the LHS (which here happens to be x).

  Python allows you to do **multiple assignments**. Try the following:

  ```
  >>> x , y = 6 , 7
  >>> x
  >>> y
  ```

  Notice that x takes the value 6 and y takes the value 7. You can simultaneously assign more than two variables if you want. Again the values in the RHS are computed *first*, and *then* these values are assigned to the variables in the LHS. More details can be found in Section 2.5.3.

- Now that we know something about variables, expressions, assignments, and types, we can start examining some of the more mysterious aspects of chaos.py.
  - Function eval(<string>) takes a single string argument value <string>, it strips it of its quotes, and treats what's left as an expression. Try the following:

    ```
    >>> x = eval("3+2*5")
    >>> x
    >>> x = eval(2+2*5)
    ```

    The first call to eval should have given the value 13 to x and the second call should have produced an error. The following example should be clear now:

    ```
    >>> x = "2+3"
    >>> result = eval(x)
    >>> result
    ```

  - Function input(<string>) will print the <string> and will wait for the user to type something; then it returns whatever has been typed as a string. Try the following (when the interpreter waits for you to type something, go ahead and type your name without quotes):

    ```
    >>> name = input("Enter your name : ")
    >>> name
    ```

    Note that variable name now has a string value that is what you've typed (in quotes because it is of the str(ing) type). Also, note that the interpreter *first* waits for you to type something, and *then* it goes ahead to assign this value to variable name. Why? *(Hint: In which order is the RHS and LHS of the assignment statement executed?)* To clarify things, try running the following lines three times: first type 10+19, then try typing Caramba! (without any quotes), and then try "Caramba!" (with the quotes). What do you get?

    ```
    >>> x = eval(input("Give me something to evaluate:"))
    >>> x
    ```

  - Function print is examined in detail in Section 2.4. To see how it behaves, try the following examples:

    ```
    >>> print("2+4")
    >>> print("The answer is", 2+4)
    >>> print()
    >>> print("My", "name", "is", 40+54)
    >>> print("My", "name", "is", 40+54, sep="BLAH")
    >>> print("My", "name", "is", end=" ")
    >>> print(40+54, end="THIS IS THE END, NO NEW LINE")
    ```

```
>>> print(40+54)
>>> print(40+54, end="THIS IS THE END, WITH NEW LINE\n")
>>> print(40+54)
```

- In loops below we are going to use function range in the form range(<expression>). It produces a sequence of <expression> consecutive integers (starting from 0). To see the sequence, we can transform it into a list (much more about lists later) using list():

```
>>> list(range(10))
>>> x = 5
>>> list(range(3*x))
```

• **Definite loops:** A definite loop is a mechanism to repeat a sequence of statements (the <body>) a predefined (definite) number of times. The general format is the following:

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

Try the following:

```
>>> for i in [0,1,2,3]:
        print(i)
```

Note that variable i takes the values in the list [0,1,2,3] one-by-one in this order. You can achieve the same result by replacing [0,1,2,3] with range(4) which creates the same sequence:

```
>>> for i in range(4):
        print(i)

>>> x = list(range(4))
>>> x
```

We needed to transform the sequence range(4) into a list with the list() function because list is a data type (just like int and str) but sequence is not. In general, range(M) produces the M numbers 0, 1, 2, ..., M-1 in this order, which you can either use them directly in a loop, or put them in a list, and then use the list:

```
>>> x = list(range(4))
>>> x
>>> for i in x:
        print(i)
```

- The built-in function range() can do many more things. Its general format is

  ```
  range(start, n, step)
  ```

  It produces a sequence of integer numbers, starting from number `start`, going up to number `n` by increments of `step`. Not all arguments have to be present. We have already seen it in the form `range(n)`:

  ```
  >>> list(range(12))
  ```

  You can use it in the form `range(start, n)`:

  ```
  >>> list(range(2,12))
  ```

  You can use it in the full form `range(start, n, step)`:

  ```
  >>> list(range(2,12,3))
  ```

  To test yourself, first guess what list of numbers `list(range(-3,12,3))` will produce, and then try it out to confirm your guess.

Practice problems

1. Try to do all the book problems.
2. Write a for-loop that prints all numbers from 10 to 100, but every 3 numbers (i.e., 10, 13, 16, etc.), and calculates their sum. Repeat, but this time start from 100 and count *down* to 10 every 3 numbers.
3. Write a for-loop that goes through the list ['3', '5', '9+6', '8/4', '5', '39', '54', '25'] (notice that it is a list of *strings*), evaluates each string to its arithmetic value, and calculates their sum.
4. Assign strings ”32” and “65” to variables x1, x2 (try to also do it with a single assignment statement). Assign x3 = x1 + '+' + x2. What's the value of x3? what's the value of `eval(x3)`?
5. Write a program that asks for two numbers x1, x2 from the user, and then uses a *for-loop* to compute and print x1+x2, x1-x2, x1*x2, x1/x2 in order. *(Hint: The for-loop must go through the list ['+', '-', '*', '/']; study again problem (3).)*
6. Try

   ```
   eval("print('this is a piece of code', 4+5)")
   ```

   What does eval() do? Try to evaluate other pieces of code.
7. How can you put a double quote (“) inside a string? A single quote (')?