logo_CDM

CSC 241 Notes  
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

## Review: Indexing strings

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 which is a colon ‘**:**’

>>> name = 'Steve'

>>> name[0]

'S'

>>> name[0:2]

'St'

>>> name[3:4]

'v'

>>> name[1:4]

'tev'

**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:

>>> a[-1]

'o'

**Exercises**: Complete the following CodeLab exercises in the Week 2 folder:

* 51760
* 51761
* 51762
* 51763
* 51764
* 51765

# Identifiers

An **identifier** is a string that serves as an identifying name for a variable, for a function, or for a keyword (i.e. a reserved language construct such as if or print). Basically, any time you need to “name something”, the name you give is called an identifier.

*Good programmers choose identifiers that help make clear what the code is doing.*

The syntax of an identifier must follow these rules: <http://docs.python.org/reference/lexical_analysis.html#identifiers>

The table below might seem confusing at this point in your progarmming careers. (It uses a syntax relating to something called 'regular expressions'). So don't spend any time worrying about it for now -- I summarize the key points just below.

**identifier** ::= ([**letter**](http://docs.python.org/reference/lexical_analysis.html#grammar-token-letter)|"\_") ([**letter**](http://docs.python.org/reference/lexical_analysis.html#grammar-token-letter) | [**digit**](http://docs.python.org/reference/lexical_analysis.html#grammar-token-digit) | "\_")\*

**letter**  ::= [**lowercase**](http://docs.python.org/reference/lexical_analysis.html#grammar-token-lowercase) | [**uppercase**](http://docs.python.org/reference/lexical_analysis.html#grammar-token-uppercase)

**lowercase**  ::= "a"..."z"

**uppercase**  ::= "A"..."Z"

**digit**  ::= "0"..."9"

**Here are the rules you must know**:

1. The first character of an identifier must be a letter or the ‘\_’ (underscore) character. However, the underscore character is typically only used in specific situations. Therefore, for now, your identifiers should always begin with a letter.
2. The remaining characters can be any letter, digit, or underscore.
3. Identifiers are case sensitive. This means that a variable called ‘name’ is completely different from a variable called ‘Name’.
   * Note: When naming a variable, by commonly adhered to programming conventions, the first character is typically lower-case.

**Be sure that you follow the above rules.**

The following are some examples to illustrate the rules:

|  |  |  |
| --- | --- | --- |
| **Potential identifier** | **Valid?** | **Why or why not?** |
| aName | Yes |  |
| \_name | Yes |  |
| 1name | No |  |
| myName50 | Yes |  |
| f5\_34 | Yes |  |
| x67\*4 | No |  |

Are myName and myname different identifiers? Why or why not?

Suppose I wanted to store a student’s score on their first assignment in a variable. What would be a good identifier for this variable?

* 1. score
  2. student\_score
  3. assignment\_score
  4. asst1\_score

I’d probably choose ‘d’. Hopefully you will think so too!

(Incidentally, if you’re feeling particularly keen, you might write it out entirely: assignment1\_score. However, long identifiers can get tedious. As long as it’s pretty clear to you and other programmers what your code is trying to do, then your identifier is fine).

Python has certain ‘keywords’ that are reserved by the language. In fact, you’ll already recognize some of them. A reserved keyword means that you should never use it as an identifier.

Note: You do not need to memorize this list. For one thing, a good text editor (such as IDLE) prints keywords in a different color such as orange.

and del from not while

as elif global or with

assert else if pass yield

break except import print

class exec in raise

continue finally is return

def for lambda try

See <http://docs.python.org/reference/lexical_analysis.html#keywords> for this list.

# Review: Assignment statements

We learned before that the ‘=’ (equal) sign is the assignment operator used to assign a name to a value.

More precisely, the assignment statement is used to assign an identifier to an object.

>>> a = 3

>>> b = 3.0

>>> c = 'hello'

**Exercise:** Draw a diagram for each of the examples above. (See page 64 of the textbook for some diagrams).

Note: Be sure to DO THIS!

It may be tempting to skip this step and the diagrams below, but things are going to become much more complicated with respect to these variables / objects. Drawing out the diagrams will be tremendously useful for helping you understand and conceptualize things as they become more complicated.

**Data Types:** As we have discussed previously, every literal in Python has a particular “data type”. This issue is very important in programming.

**Useful function: type()**

You can use the built-in function type()to determine the data type of the value stored inside a variable. Let’s try it on the variables we declared just above:

>>> type(a)

<class 'int'>

>>> type(b)

<class 'float'>

>>> type(c)

<class 'str'>

>>> t = False

>>> type(t)

<class 'bool'>

**How does Python do assignments**?

1. Before an identifier is assigned a value, it does not yet exist.
2. The variable only exists when it is assigned a value for the first time. For example, if we say **x = 3**, the variable ‘x’ is created. Then subsequent assignments to ‘x’ (such as x = ‘hello’) will use the existing identifier x.
3. Therefore, the same identifier can refer to different data types at different times. In the example above, x firsts refers to an integer (3) and then later in the same program to a value of another data type, in this case, a String.
4. When a variable appears in an expression, it is immediately evaluated (i.e. replaced by its value). This also means that the use of unassigned variables (i.e. variables that have no value) in an expression is an error as seen below:

>>> print(d+4)

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

d+4

NameError: name 'd' is not defined

As you see, the error you get when you try to reference an un-assigned variable is: **undefined**

**Practice problems**:

*Again, please do not skip this exercise. Be sure that you do these diagrams.*

* Draw a diagram representing the state of names and objects after the following execution:

>>> a = 4444

>>> b = a

* Draw a diagram representing the state of names and objects after the following execution:

>>> a = 4444

>>> b = a

***IMPORTANT: We will do this next one togther in class, because Python does something very unusual with integers:***

>>> a = 5555

**It is very important that you follow what just happened here. However, we will review it at least 1 or 2 more times.**

# Comments and docstrings

Good programmers write code that is clear and relatively easy to follow. One extremely useful tool that all good programmers use is to include “comments” in their code.

Comments are snippets of text that is ignored by the Python in iterpreter. It is a way for you as the progarmmer to explain and clarify what your code is doing.

*Python programs should be well documented using comments.* Remember that one overarching goal of your code is to make it clear. Commenting is, of course, a great way to do that!

Documentation is important because undocumented code is harder to maintain, even by the programmer who wrote the code.

Placing a pound sign (aka hash tag) tells the Python interpreter that everything that follows on the rest of the line is a comment.

The following shows an example program found in the file **triangle.py** that includes documentation:

# A program that checks whether a sequence of numbers

# a, b, and c input by the user satisfy the equation

# a^2 + b^2 = c^2

a = eval(input("Enter the shortest side: "))

b = eval(input("Enter the next shortest: "))

c = eval(input("Enter the longest side: "))

if (a\*\*2 + b\*\*2) == c\*\*2:

print("Right triangle") # a^2+b^2 == c^2

else:

print("Oblique triangle") #a^2 + b^2 != c^2

We’ll learn more about the code at the end of this program soon.

# Lists

**Lists** are ordered collections of objects.

*You wil use Lists and related structures a lot in any field that uses programming. So make sure you practice and get good at working with them!*

Lists have the following **properties**:

* The objects are ordered left to right
* Each object in the list is accessible with an index (just like with strings), 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 be modified. In programming jargon we say that they are “**mutable**”. Conversely, we say that strings are "immutable".
* Lists have dynamic lengths, meaning that they can grow and shrink as items are added to, or deleted from the list

The following demonstrates some simple list operations.

To create a list, we use square brackets:

>>> lst1 = [] #this is an empty list

>>> lst2 = [1, 2, 3, 5, 7, 11, 13] #a list with 7 integers

>>> lst3 = [lst2, 2, 4, 6, ["hello", "goodbye"]]

#Note that the first item in this list is another list!

>>> lst3

[[1, 2, 3, 5, 7, 11, 13], 2, 4, 6, ['hello', 'goodbye']]

>>> len(lst1)

0

>>> len(lst2)

7

>>> len(lst3)

5

>>> lst2 + lst2 #Concatenating 2 lists together

[1, 2, 3, 5, 7, 11, 13, 1, 2, 3, 5, 7, 11, 13]

>>> lst2 \* 3

[1, 2, 3, 5, 7, 11, 13, 1, 2, 3, 5, 7, 11, 13, 1, 2, 3, 5, 7, 11, 13]

**A very useful command** when working with lists is the ‘**in**’ command. This returns true if the first argument is present inside the second argument:

>>> 2 in lst2

True

>>> 4 in lst2

False

NOTE: In the examples above, the lists hold values of different data types. However, you’ll find that most of the time, we work with lists that contain values of only a single data type (e,g, all values in the list are integers, or floats, or Strings, etc)

## Indexing/Slicing Lists

As with strings, we can do indexing and slicing on lists:

>>> lst2[2] #indexing

3

>>> lst2[2:5] #slicing

[3, 5, 7]

>>> lst2[-1]

13

## Modifying Lists

These examples demonstrate how to change lists:

>>> lst2

[1, 2, 3, 5, 7, 11, 13]

>>> lst2**[5]** = 17

>>> lst2

[1, 2, 3, 5, 7, 17, 13]

>>> lst2.**append(17)**

>>> lst2[5] = 11

>>> lst2

[1, 2, 3, 5, 7, 11, 13, 17]

>>> lst2.**reverse()**

>>> lst2

[17, 13, 11, 7, 5, 3, 2, 1]

>>> lst2.**sort()**

>>> lst2

[1, 2, 3, 5, 7, 11, 13, 17]

>>> **del** lst2[3]

>>> lst2

[1, 2, 3, 7, 11, 13, 17]

>>> del lst2[2:4]

>>> lst2

[1, 2, 11, 13, 17]

**Mutability of Lists / Immutability of Strings:**

The command

lst2[0] = 25

will successfully change the first item in lst2 to the value 25.

However, suppose we have the string:

name="Roberta"

If we tried to say

name[0] = 'B'

We would get an error! This is because strings are immutable.

The only way to change the string would be to recreate it with the new value:

name = 'Boberta"

**CodeLab exercises**: Complete the following problems in the Week 2 folder.

Hint: Feel free to use Python's help() function as **help(list)** to see some functions that might be useful to you.

* 51194
* 51199
* 51200
* 51201
* 51603
* 51606

**Exercise**: Given the list of student homework grades

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

Write Python expressions to:

* Find the number of grades with a score of 7
* Change the value of the last grade to 4
* Sort the grades list
* Find the highest grade (i.e. sort first)
* Find the average of the three lowest grades (i.e. sort first)

**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.