Intro to Python

Feb 25, 2018

Contents

- Why Python?
- Types, Variables and Operators
- Input/Output
- Strings
- Branching
- Iteration
- Functions
- Data Structures
- Python in.. (Libraries)

Why Python?

Python is a living language. Since its introduction by Guido von Rossum in 1990, it has undergone many changes. Earlier it was little known and used language, but with the arrival of Python 2.0 in 2000 it changed

- Popular
- Open Source
- Cross Platform
- Easy to learn
- Readable Code
- General Purpose - used for games to robotics to data analysis

NOTE: For this workshop we will be using python 3.x

Basic elements of Python

Python is **interpreted** and a Python program, sometimes called a **script**, is a sequence of definitions and commands. These definitions are evaluated and the commands are executed by the **Python interpreter** in something called the **shell**.

Basic elements of Python

We recommend you start a Python shell now and use it to try examples that follow.

Open Terminal (Shortcut: ctrl + alt + T)

Type python3 (after student@student-ubuntu : ~$)

Now you are using python interpreter. Try using these commands:

```
print ('Yankees rule!')

print ('But not in Boston!')

print ('Yankees rule,', 'but not in Boston!')
```

Types, Variables and Operators

Objects are the core things that Python programs manipulate. Every object has a **type** that defines the kinds of things programs can do with those type of objects.

Types are either **Scalar** or **Non-Scalar**. Scalar are indivisible(like atoms)**.** Non-Scalar have their own internal structure.

Python has four type of scalar objects:

- int - to represent integer values
- float - to represent floating point numbers
- bool - to represent True and False
- None - It is unique to python and it represents a single value

You can use type() to check the type of the object. Example: type(5) -> int

Types, Variables and Operators

Sometimes we want to change the type of an object especially with numbers. This is called **Casting**.

This way you can convert object from one type to another.

Try:

float(3) -> 3.0

int(3.9) -> 3 (truncates)

Types, Variables and Operators

Objects and operators can be combined to form expressions, each of which evaluates to an object of some type.

For example, the expression 3 + 2 denotes the object 5 of type int, and the expression 3.0 + 2.0 denotes the object 5.0 of type float.

Now the basic operators in python are as same as any other programming language i.e

- Arithmetic Operators [+, -, *, /, %, **, //]
- Relational Operators [==, !=, <, >, >=, <=]
- Assignment Operators [=, +=, -=, *=, /=, %=, **=]
- Logical Operators [and, or, not]
- Membership Operators [in, not in]

Types, Variables and Operators

Variables provide a way to associate names with objects.

Example: pi = 3, radius = 11

Tip: Apt choice of variable names plays an important role in enhancing readability. Consider the two code fragments:

```
a = 3.14159
b = 11.2
c = a*(b**2)
```

```
pi = 3.14159
radius = 11.2
area = pi*(radius**2)
```

Types, Variables and Operators

In Python, variable names can contain uppercase and lowercase letters, digits (but they cannot start with a digit), and the special character _. Python variable names are case-sensitive e.g., Julie and julie are different names. Finally, there are a small number of reserved words (sometimes called keywords) in Python that have built-in meanings and cannot be used as variable names.

Input/Output

Now input provided by the user is very important, in Python 3, we use **input()** function to get input directly from the user. This takes input as string which can be type casted to another types. To print a specific input message we write that message in the parenthesis of input().

Examples:

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

Enter your name: Jon Snow

>>> print(name)

Jon Snow
```

Input/Output

>>> print(‘Are you really’, name, ‘?’)

Are you really Jon Snow ?

>>> print(‘Are you really’, name + ‘?’)

Are you really Jon Snow?

Notice that the first print statement introduces a blank before the “ ? ” It does this because when print is given multiple arguments it places a blank space between the values associated with the arguments. The second print statement uses concatenation to produce a string that does not contain the superfluous blank and passes this as the only argument to print.

Input/Output

Since input() takes input as string. Consider,

>>> n = input('Enter an int: ')

Enter an int: 3

>>> print type(n)

<type 'str'>

Notice that the variable n is bound to the str '3' not the int 3 . So, for example,

the value of the expression n*4 is '3333' rather than 12. Thus it needs to be explicitly converted to int by using int(n)

Strings

Objects of type str are used to represent strings of characters. 11 Literals of type str can be written using either single or double quotes, e.g., 'abc' or "abc" .

The literal '123' denotes a string of characters, not the number one hundred twenty-three.

Try typing the following expressions in to the Python interpreter (remember that the >>> is a prompt, not something that you type):

Strings

```
>>> 'a'
>>> 3*4
>>> 3*'a'
>>> 3+4
>>> 'a'+'a'
```

The operator + is said to be overloaded: It has different meanings depending upon the types of the objects to which it is applied.

Strings

Strings are one of several sequence types in Python. They share the following operations with all sequence types.

- The **length** of a string can be found using the len function. For example, the value of len('abc') is 3.

- **Indexing** can be used to extract individual characters from a string. In Python, all indexing is zero-based. For example, typing 'abc'[0] into the interpreter will cause it to display the string 'a'. Typing 'abc'[3] will produce the error message IndexError: string index out of range. Negative numbers are used to index from the end of a string. For example, the value of 'abc'[-1] is 'c'.

Strings

- **Slicing** is used to extract substrings of arbitrary length. If s is a string, the expression s[start:end] denotes the substring of s that starts at index start and ends at index end-1 . For example, 'abc'[1:3] = 'bc'. If the value before the colon is omitted, it defaults to 0 . If the value after the colon is omitted, it defaults to the length of the string. Consequently, the expression 'abc'[:] is semantically equivalent to 'abc'.

Some string library functions:

string.ascii_letters - gives you combination of both lower and upper case letters.

String.digits - gives you the string ‘0123456789’.

Finger Exercise

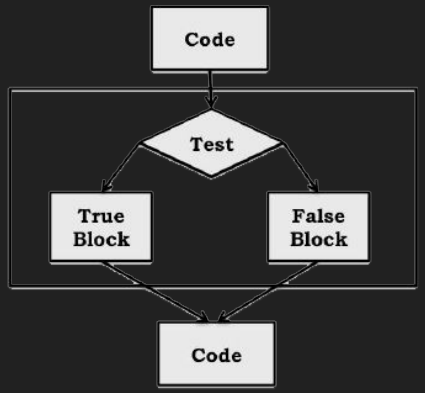
- Write a program to count the vowels inside a string. It should check both lowercase and uppercase characters.

Branching

The kinds of computations we have been looking at thus far are called straight-line programs. They execute one statement after another in the order in which they appear, and stop when they run out of statements.

Branching programs are more interesting. The simplest branching statement is a conditional.

Branching



As depicted in Figure, a conditional statement has three

parts:

- a test, i.e., an expression that evaluates to either True or False ;
- a block of code that is executed if the test evaluates to True ; and
- an optional block of code that is executed if the test evaluates to False .

After a conditional statement is executed, execution resumes at the code following the statement.

Branching

In Python, a conditional statement has the form:

if Boolean expression:

    block of code

else:

    block of code

Tip: Pay special attention to the indentation.

Branching

Consider the following program that prints “Even” if the value of the variable x is even and “Odd” otherwise:

```
if x%2 == 0:
        print ('Even')
else:
        print ('Odd')
print ('Done with conditional')
```

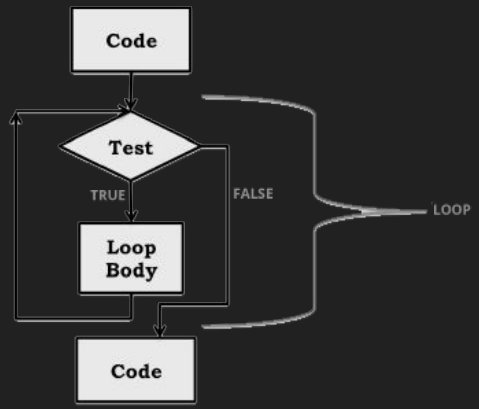
Finger Exercise

Finger exercise: Write a program that examines three variables— x , y , and z — and prints the largest odd number among them. If none of them are odd, it should print a message to that effect.

Iteration

The third type of flow of control is iteration (or looping). Imagine you had a job which you have to do repeatedly for multiple times, now its not a good programming exercise to write the same code that many times instead we can have a loop.

Like a conditional statement it begins with a test. If the test evaluates to True , the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False , after which control passes to the code following the iteration statement.



Iteration (Example)

```
# Square an integer, (while loop)

x = 3

ans = 0

itersLeft = x

while (itersLeft != 0):

        ans = ans + x

        itersLeft = itersLeft - 1

print (str(x) + '*' + str(x) + ' = ' + str(ans))
```

Iteration

Syntax of while loop

```
test_variable = value          #declaring control variable
while(test_condition):                    #checking against the condition
        code to be repeated
        test_variable += 1       #update the control variable
#statement not part of while
```

Iteration

Syntax of for loop:

for *variable* in *sequence*:

    code block

The variable following for is bound to the first value in the sequence, and the code block is executed. The variable is then assigned the second value in the sequence, and the code block is executed again. The process continues until the sequence is exhausted or a break statement is executed within the code block.

Example:

```
for i in range(0, 4):

    print (i)
```

It prints:
0
1
2
3

Functions

In Python each function definition is of the form.

```
def name of function (list of formal parameters):
    body of function
```

For example, we could define the function max18 by the code

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Functions

def is a reserved word that tells Python that a function is about to be defined. The function name (max in the last example) is simply a name that is used to refer to the function.

The sequence of names (x,y in this example) within the parentheses following the function name are the formal parameters of the function. When the function is used, the formal parameters are bound (as in an assignment statement) to the actual parameters (often referred to as arguments) of the function invocation (also referred to as a function call). For example, the invocation

max(3, 4) binds x to 3 and y to 4.

Finger exercise: Write a function isIn that accepts two strings as arguments and returns True if either string occurs anywhere in the other, and False otherwise. Hint: you might want to use the built-in str operation in.

Functions

In Python, there are two ways that formal parameters get bound to actual parameters. The most common method, which is the only one we have used thus far, is called positional—the first formal parameter is bound to the first actual parameter, the second formal to the second actual, etc. Python also supports what it calls keyword arguments, in which formals are bound to actuals using the name of the formal parameter. Example

```
def printName(firstName,  lastName, reverse):
        if reverse:
                print(lastName,",", firstName)
        else:
                print(firstName, lastName)
```

Functions

Each of the following is an equivalent invocation of printName:

printName('Olga', 'Puchmajerova', False)

printName('Olga', 'Puchmajerova', False)

printName('Olga', 'Puchmajerova', reverse = False)

printName('Olga', lastName = 'Puchmajerova', reverse = False)

printName(lastName='Puchmajerova', firstName='Olga', reverse=False)

Functions

Keyword arguments are commonly used in conjunction with default parameter values. We can, for example, write

```
def printName(firstName, lastName, reverse = False):
    if reverse:
        print lastName + ', ' + firstName
    else:
        print firstName, lastName
```

Default values allow programmers to call a function with fewer than the specified number of arguments.

Finger Exercise

Write a python function that takes an integer x as input and returns the value of x raised to fourth power.

Recursion

You may have heard of recursion, That’s an urban legend spread by computer scientists to make people think that we are smarter than we really are. Recursion is a very important idea, but it’s not so subtle.

In simple terms, it is the process of calling itself from the function.

In general, a recursive definition is made up of two parts. There is at least one base case that directly specifies the result for a special case (case 1 in the example above), and there is at least one recursive (inductive) case (cases 2 and 3 in the example above) that defines the answer in terms of the answer to the question on some other input, typically a simpler version of the same problem.

Recursion

Consider an example of recursion: Program to calculate factorial of a number.

The classic **inductive definition** is,

1! = 1

(n + 1)! = (n + 1) * n!

The first equation defines the base case. The second equation defines factorial for all natural numbers, except the base case, in terms of the factorial of the previous number.

Recursion

ITERATIVE VERSION

```
def factI(n):
    """Assumes that n is an int > 0
    Returns n!"""
    result = 1
    while n > 1:
        result = result * n
        n -= 1
    return result
```

RECURSIVE APPROACH

```
def factR(n):
    """Assumes that n is an int > 0
    Returns n!"""
    if n == 1:
        return n
    else:
        return n*fact(n - 1)
```

Finger Exercise

- Write a function recurPower(base, exp) which computes base^exp by recursively calling itself to solve a smaller version of the same problem, and then multiplying the result by base to solve the initial problem.

Data Structures (Lists)

The list is the most versatile datatype available in Python, which can be written a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

list1 = [‘physics’, ‘chemistry’, 1997, 2000]

list2 = [1, 2, 3, a, b]

Data Structures (Lists)

- **Accessing Values in Lists**: To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.
  For example −
  >>>print(list1[0])
  physics
  >>>print(list2[1:5])
  [2, 3, a, b]

- **Delete List Elements**: To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting. You can use the remove() method if you do not know exactly which items to delete.
  For example –
  list1 = [‘physics’, ‘chemistry’, 1997, 2000]
  >>>print(list1)
  >>>del(list1[2])
  >>>print(list1)
  >>>print(list1)
  >>>list1.remove(‘chemistry’)
  >>>print(list1)

Data Structures (Lists)

Basic List Operations:

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end = ' ')	1 2 3	Iteration

Data Structures (Lists)

Basic List Operations:

- **list.append(x)** - Adds an item to the end of the list.
- **list.extend(*iterable*)** - extends the list by appending all the items from the iterable.
- **list.insert(i, x)** - inserts an item x at ith place in the list
- **list.clear()** - removes all the elements from the list
- **list.index(x, start, end)** - returns zero-based index in the list of the first element that is equal to x. Raises ValueError otherwise.
- **list.count(c)** - returns the number of times x appears in the list
- **list.sort(key = None, reverse=None)** - sorts the list in place
- **list.reverse()** - reverses the list in place.

Data Structures (Tuple)

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between tuples and lists is that tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values.

Tup1 = (‘physics’, ‘chemistry’, 1997, 2001)

Tup2 = (1, 2, 3, ‘a’ ,’b’)

Like string indices, tuples indices starts at 0, and they can be sliced concatenated etc.

Finger exercise: try same operations like updating one index and deleting tuple as you did for lists

Data Structures (Dictionary)

Dictionary is basically a “key : value” pair in Python. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Data Structures (Dictionary)

Accessing values in Dictionary: To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

>>>print ("dict['Name']: ", dict['Name'])

dict[‘Name’]: Zara

>>>print ("dict['Age']: ", dict['Age'])

dict[‘Age’]: 7
```

If we attempt to access a data item with a key, which is not a part of the dictionary, we get an error.

Data Structures (Dictionary)

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry
```

Data Structures (Dictionary)

Delete Dictionary Elements
You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the del statement. Following is a simple example −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name'] # remove entry with key 'Name'
dict.clear()     # remove all entries in dict
del dict         # delete entire dictionary
```

An exception is raised because after del dict, the dictionary does not exist.

Data Structures (Dictionary)

Some Dictionary Operations:

- **dict.clear()** - Removes all elements of dictionary dict.
- **dict.get(key, default=None)** - For key *key*, returns value or default if key not in dictionary.
- **dict.items()** - Returns a list of dict's (key, value) tuple pairs
- **dict.keys()** - Returns list of dictionary dict's keys
- **dict.values()** - Returns list of dictionary dict's values
- **dict.update(dict2)** - Adds dictionary dict2's key-values pairs to dict

Finger Exercise

Consider the following sequence of expressions:

animals = {‘a’: [‘aardavak’], ‘b’: [‘baboon’], ‘c’: [‘coati’]}

animals[‘d’] = [‘donkey’]

animals[‘d’].append(‘donkey’)

animals[‘d’].append(‘dingo’)

Write a procedure, called how_many, which returns the sum of the number of values associated with a dictionary. For example:

>>>print(how_many(animals))

6

Data Structures (Sets)

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection and difference.

Curly braces or the set() function can be used to create sets.

Example:

>>>basket = {‘apple’, ‘orange’, ‘pear’, ‘orange’, ‘apple’}

>>>print(basket) **#shows duplicates are removed**

{‘apple’, ‘orange’, ‘pear’}

Data Structures (Sets)

Some set operations on unique letters from two words:

```
>>>a = set('abracadabra')

>>>b = set('alacazam')

>>>a

{'a', 'b', 'r', 'c', 'd'}

>>>a - b

{'r', 'd', 'b'}
```

```
>>>b

{'a', 'l', 'c', 'z', 'm'}
```

Data Structures (Sets)

>>>a | b **#represents a union b**

{'r', 'a', 'd', 'l', 'z', 'm', 'c', 'b'}

>>> a & b **#represents a intersection b**

{'c', 'a'}

>>>a ^ b **#returns letters in a or b but not both**

{'r', 'l', 'd', 'z', 'm', 'b'}

Python in.. (Libraries)

- Web
  - Django, ZOPE, WebPy, TurboGears
- Scientific Computing
  - NumPy, SAGE
- GUI Development
  - PyGtk, PyQt
- Game Dev
  - PyGame
- Network Programming
  - Twisted
- Graph
  - NetworkX