Programming Lab

Autumn Semester

Course code: PC503

I can toast!
Let me rephrase that
I can do ANYTHING you tell me to...
... provided you speak my language

Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 5

Different Data Types

Lists

- Python knows a number of *compound* data types, used to group together other values.
- The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets.
- Lists might contain items of different types, but usually, the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

- Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0]
1
>>> squares[-1]
25
>>> squares[-3:]
[9, 16, 25]
```

Lists

- All slice operations return a new list containing the requested elements.

- This means that the following slice returns a shallow copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

- Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # Something's wrong here
>>> 4 ** 3  # The cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Lists

- You can also add new items at the end of the list, by using the append() *method* :

```
>>> cubes.append(216)  # Add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

- Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> letters[:] = []
>>> letters
```

Lists

- The built-in function len() also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

- It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
>>>
```

First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
>>> # The sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

- The first line contains a multiple assignment: the variables **a and b** simultaneously get the new values **0 and 1**.

- On the last line, this is used again, demonstrating that the expressions on the ***right-hand side are all evaluated first before any of the assignments take place.*** The right-hand side expressions are evaluated from the left to the right.

- The while loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any ***non-zero integer value is true; zero is false.***

- The condition may also be a string or list value, in fact, any sequence; anything with a non-zero length is true, and empty sequences are false.

- The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: ***< (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to), and != (not equal to).***

- The body of the loop is indented: ***indentation is Python’s way of grouping statements.***

- At the interactive prompt, you have to type a tab or space(s) for each indented line.

- In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility.

- When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (***since the parser cannot guess when you have typed the last line***).

- Note that each line within a basic block must be indented by the same amount.

- The **print()** function writes the value of the argument(s) it is given.

- It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings.

- Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
>>>
```

- The keyword argument end can be used to avoid the newline after the output, or end the output with a different string:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,>>>
```

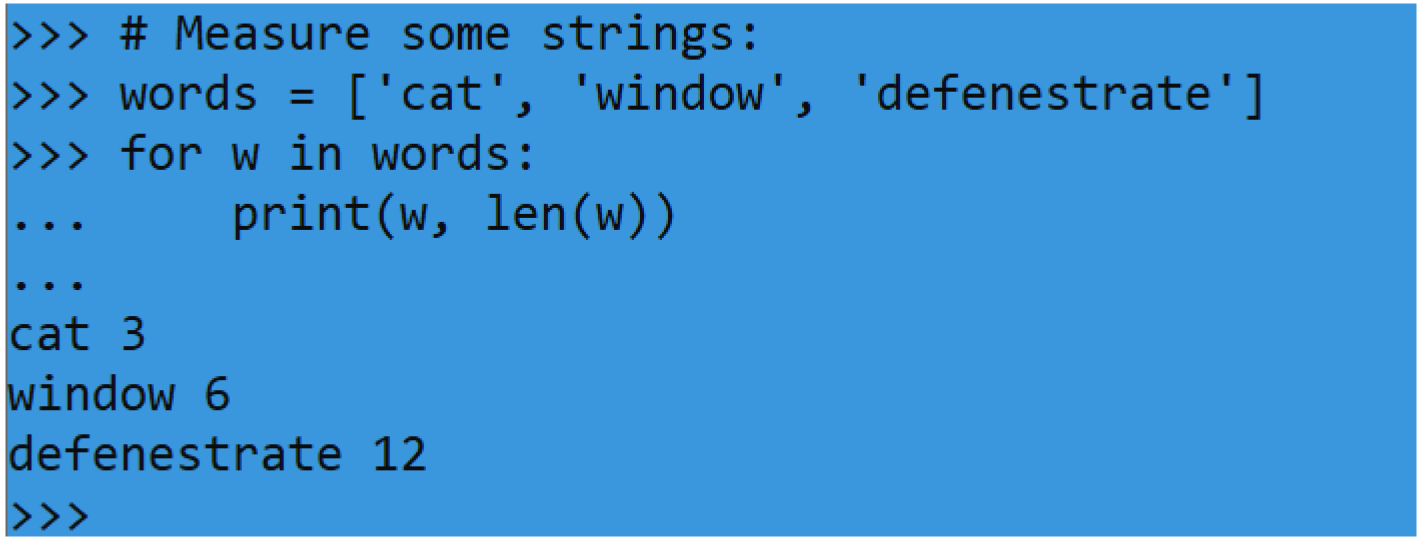
More Control Flow Tools

if Statements

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 34
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
>>>
```

More Control Flow Tools

for Statements



```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
>>>
```

More Control Flow Tools

- Code that modifies a collection while iterating over that same collection can be tricky to get right.
- Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
>>> # Create a sample collection
>>> users = {'rahul': 'active', 'programming': 'inactive',
'course': 'active'}
>>> # Strategy:  Create a new collection
>>> for user, status in users.copy().items():
...     if status == 'inactive':
...         del users[user]
...
>>> users
{'rahul': 'active', 'course': 'active'}
>>>
>>> # Strategy:  Create a new collection
>>> active_users = {}
>>> for user, status in users.items():
...     if status == 'inactive':
...         active_users[user] = status
...
>>> users
{'rahul': 'active', 'course': 'active'}
>>>
```

More Control Flow Tools

- **The range() Function**

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))
[-10, -40, -70]
>>>
```

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
>>>
```

More Control Flow Tools

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

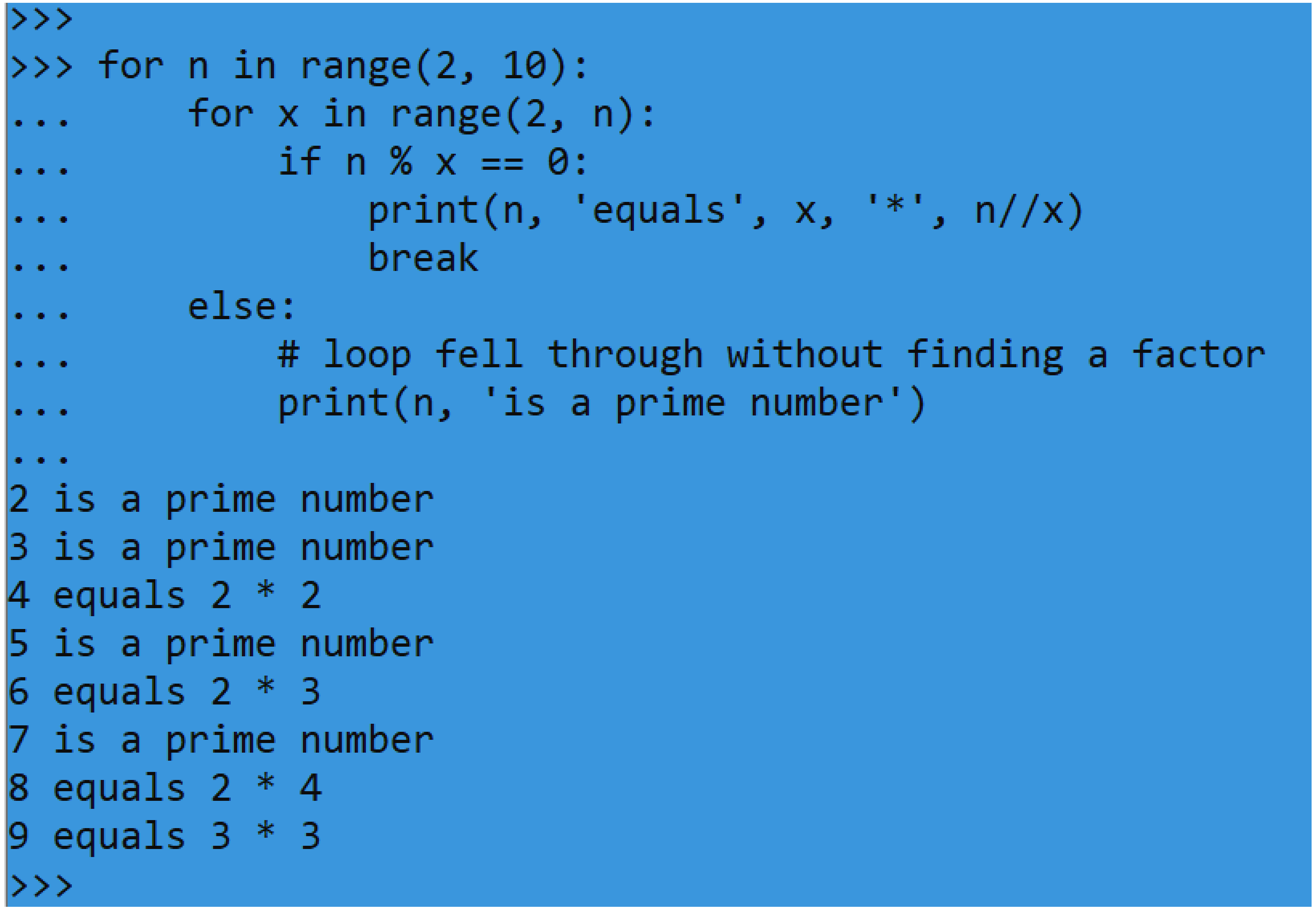
In many ways the object returned by range() behaves as if it is a list, but in fact it isn’t.

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
>>>
```

break and continue Statements, and else Clauses on Loops

- The **break** statement breaks out of the innermost enclosing for or while loop.
- A **for or while** loop can include an else clause.
- *In a for loop, the else clause is executed after the loop reaches its final iteration*.
- *In a while loop, it's executed after the loop's condition becomes false*.
- In either kind of loop, the else clause is not executed if the loop was terminated by a break.

- for loop, which searches for prime numbers:



```
>>>
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
>>>
```