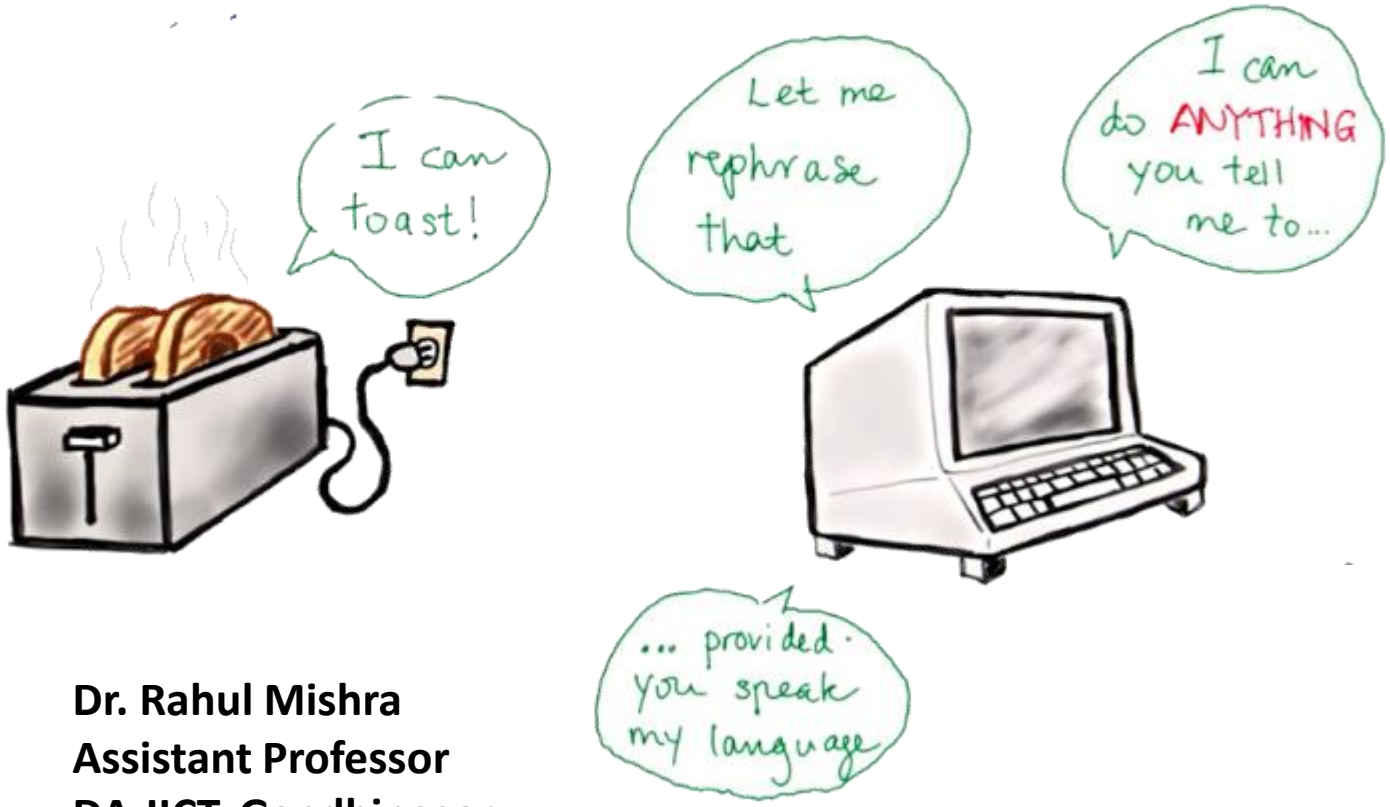
Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 2

Strings onwards

An Informal Introduction to Python

In addition to int and float, Python supports other types of numbers, such as ***Decimals and fractions***.

Python also has built-in support for complex numbers and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

Strings

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways.
- They can be enclosed in single quotes ('...') or double quotes ("...") with the same result
- \ can be used to escape quotes:

```
>>> 'programming course'
'programming course'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

An Informal Introduction to Python

```
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> \"Yes,\" they said.
  File "<stdin>", line 1
    \"Yes,\" they said.
     ^
SyntaxError: unexpected character after line continuation character
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> '"Isn't," they said.'
  File "<stdin>", line 1
    '"Isn't," they said.'
          ^
SyntaxError: invalid syntax
>>> " "Yes," they said."
  File "<stdin>", line 1
    " "Yes," they said."
       ^^^
SyntaxError: invalid syntax
>>>
```

An Informal Introduction to Python

- In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes.

- While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent.

- The *string is enclosed in double quotes* if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes.

- The ***print()*** function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

An Informal Introduction to Python

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

- String literals can span multiple lines
- One way is using triple-quotes: """..."""" or '''...'''
- End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line.

```
Microsoft Windows [Version 10.0.19045.3208]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator>python
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:37) [MSC v.1934 64 bit (AMD64)] on
 win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("""\
... Usage: thingy [OPTIONS]
...        -h                        Display this usage message
...        -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
>>>
```

1. Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.
2. This feature is particularly useful when you want to break long strings:

```
>>> 'Py' 'thon'
'Python'
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
>>>
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'  # can't concatenate a variable and a string literal
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
>>>
```

If you want to concatenate variables or a variable and a literal, use +:

```
>>> prefix + 'thon'
'Python'
>>>
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python PC 503'
>>> word[0]
'P'
>>> word[5]
'n'
>>> word[-1]
'3'
>>> word[-5]
'C'
>>>
```

Using similar technique try to print in reverse “Programming Course 2023”

Solution: “Programming Course 2023”

```
>>> word = "Programming Course 2023"
>>> word
'Programming Course 2023'
>>> for i in range(-1,-23,-1):
...     print(word[i])
...
3
2
0
2

e
s
r
u
o
C

g
n
i
m
m
a
r
g
o
r
>>>
```

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on PC 503'
>>> word[4:]   # characters from position 4 (included) to the end
'on PC 503'
>>> word[:2] + word[2:]
'Python PC 503'
>>> word[:4] + word[4:]
'Python PC 503'
>>>
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> word[4:42]
'on PC 503'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

1. If you need a different string, you should create a new one:

2. The built-in function len() returns the length of a string:

```
>>> 'J' + word[1:]
'Jython PC 503'
>>> word[:2] + 'py'
'Pypy'
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
>>>
```