Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 4
Working with Text

Strings, String Operations, Printing Information, Reading from the Keyboard

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

Abstract

"Nothing is so obvious that it's obvious… The use of the word "obvious" indicates the absence of a logical argument."

– Errol Morris

Text plays a central role in computer programs.
We will introduce a non-numeric data type that represents text.
We will see how to make programs more interactive by printing messages and getting information from the user.

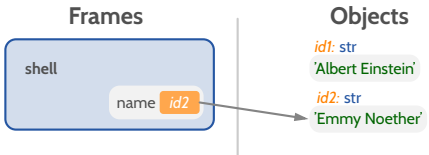
Overview

- Computers have been invented to do arithmetic
- Now they spend a lot (most?) of their time processing text
- Email, web browsers, word processors…
- Text input & output (I/O) allow for user interaction
- In Python text is represented as *strings*
- Strings are sequences of *characters*

Manipulating text is an important part of programming.

String Examples

```
>>> 'Albert Einstein'
'Albert Einstein'
>>> name = "Emmy Noether"
>>> name
'Emmy Noether'
```

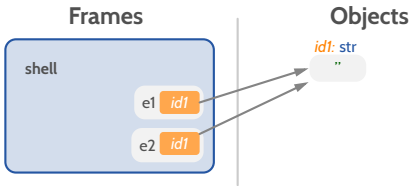


- Python *string literals* are characters surrounded by *quotes*; they are also expressions
- Python strings have the type `str`
- The quotes can be single or double quotes
- Use what you prefer
- But please use double quotes for *docstrings*
- Strings can be assigned to variables

Characters are letters, digits, punctuation and so on.

String Facts

```
>>> e1 = ''
>>> e1
''
>>> e2 = ""
>>> e2
''
```



- Strings can be *empty*
- Python strings can have arbitrary length (within memory limits)

Note that Python caches the empty string.

String Quoting Rules

- Quotes must match

```
>>> 'Charles Darwin"
  File "<stdin>", line 1
    'Charles Darwin"
                   ^
SyntaxError: EOL while scanning string literal
```

- If you need quotes inside a string, use the other quotes to surround (*delimit*) the string

```
>>> 'that's not going to work'
  File "<stdin>", line 1
    'that's not going to work'
         ^
SyntaxError: invalid syntax
```

```
>>> "that's much better!"
'that's much better!'
```

Notice the weird coloring in the second example. What might cause this?

String Length

- We often need to know the length of a string
- The builtin function `len` give us just that:

```
>>> len('Jane Goodall')
12
>>> len('123!')
4
>>> len(' ')
1
>>> len('')
0
```

- The length is the number of characters between the quotes
- The empty string has length `0`

Note that a string with only spaces is *not* empty.

String Concatenation

```
>>> 'Jane' + ' Goodall'
'Jane Goodall'
>>> 'Margaret Hamilton' + ""
'Margaret Hamilton'
```

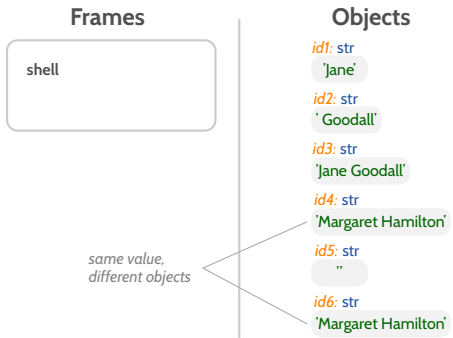
- It is common to *concatenate* strings
- In python this is done with the + operator
- Concatenation creates a new object

The + operator is meaningful for many types. We'll see more examples later.

String Concatenation

```
>>> 'Jane' + ' Goodall'
'Jane Goodall'
>>> 'Margaret Hamilton' + ""
'Margaret Hamilton'
```

- It is possible concatenate a string with the empty string
- This yields a string with the same value



Note there are now two *distinct* objects with the same value.

String Concatenation

- What happens when we try to add a number to a string?

```
>>> 'NH' + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

- The error message depends on the order of the operands

```
>>> 8 + ' planets'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python can't figure out what we want to do here.

String Concatenation

- Use the builtin `str` function to convert a number to a string:

```
>>> 'There are ' + str(4) + ' lights!'
'There are 4 lights!'
```

```
>>> 'One mile is ' + str(1.60934) + ' km.'
'One mile is 1.60934 km.'
```

The concatenation operator must be applied to two strings.

String Conversions

- If a string looks like an integer, you can convert it to an integer using the builtin function `int`:

```
>>> int('0')
0
>>> int('13')
13
>>> int('-721')
-721
```

- For floating point variables we use the function `float`:

```
>>> float('-721')
-721.0
>>> float('47.1')
47.1
```

This is extremely useful when reading in data.

String Conversions

- This does not work for all strings:

```
>>> int('thirteen')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'thirteen'
```

```
>>> float('pi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'pi'
```

That's a problem when dealing with user input. We'll learn how to handle it soon.

Repeating Strings

- We can repeat strings using the `*` operator:

```
>>> 'ATT' * 5
'ATTATTATTATTATT'
>>> '-' * 8
'--------'
```

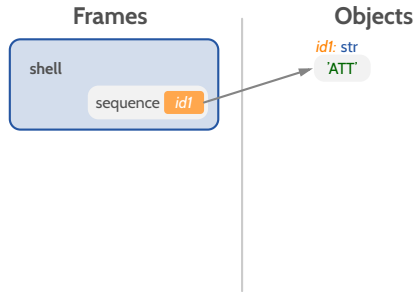
- Repeating with `0` or a negative number yields the empty string:

```
>>> 'GC' * 0
''
>>> 'TATATATATA' * -3
''
```

Speak precisely: this is *repetition*, *not* multiplication.

String Expressions

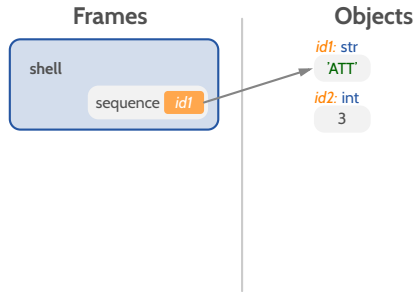
```
>>> sequence = 'ATT'
>>> len(sequence)
3
>>> new_sequence = sequence + 'GGC'
>>> new_sequence *= 2
```



- String literals can be assigned to variables

String Expressions

```
>>> sequence = 'ATT'
>>> len(sequence)
3
>>> new_sequence = sequence + 'GGC'
>>> new_sequence *= 2
```

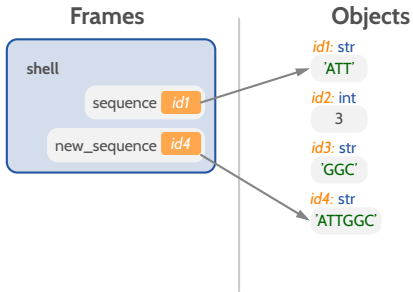


- String literals can be assigned to variables
- The `len` function yields information about a string object but doesn't change it

String Expressions

```
>>> sequence = 'ATT'
>>> len(sequence)
3
>>> new_sequence = sequence + 'GGC'
>>> new_sequence *= 2
```

- String literals can be assigned to variables
- The `len` function yields information about a string object but doesn't change it
- Expressions with string results always create new objects

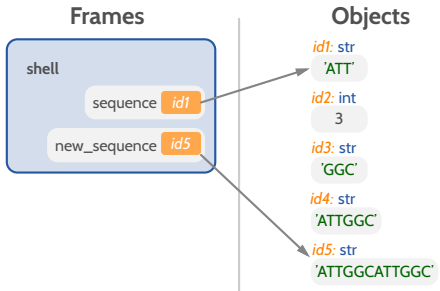


String Expressions

```
>>> sequence = 'ATT'
>>> len(sequence)
3
>>> new_sequence = sequence + 'GGC'
>>> new_sequence *= 2
```

- String literals can be assigned to variables
- The `len` function yields information about a string object but doesn't change it
- Expressions with string results always create new objects
- This also applies to augmented assignment operators

Strings, just like numbers, are *immutable*.



Special Characters

- What if you want single and double quotes in one string? You could do this:

```
>>> 'She said, "That' + "'" + 's hard to read."'
'She said "That\'s hard to read."'
```

- The backslash `\` is an *escape character*
- `\'` is an *escape sequence*
- Escape sequences have length 1:

```
>>> len('that\'s')
6
```

Escape sequences are useful for formatted output.

Escape Sequences

Escape Sequence	Description
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\t`	tab
`\n`	newline
`\r`	carriage return

Multiline Strings

- Strings defined with single or double quotes must be defined on one line:

```
>>> 'one:
  File "<stdin>", line 1
    'one
       ^
SyntaxError: EOL while scanning string literal
```

- We can define multiline strings using three quotes:

```
>>> """one
... two
... three"""
'one\ntwo\nthree'
```

Note the newline escape sequences in the representation.

Normalizing Newlines

- Different operating systems use different characters to indicate the end of a line:

System	Sequence
Mac OSX & Linux/Unix	`\n`
Windows	`\r\n`
Old Mac OS	`\r`

- Python *normalizes* newlines
- It always uses `\n` to indicate a new line
- Python programmers can write the same code for all systems

Printing Information

- We use the builtin function `print` to show messages to the users of our programs:

```
>>> print(1 + 1) # printing the result of an expression
2
>>> print("You can't take the sky from me.") # printing a string
You can't take the sky from me.
```

- Note that the quotes are stripped when printing a string
- The `print` function renders escape sequences instead of showing them:

```
>>> print('one\ttwo\nthree\tfour')
one     two
three   four
```

The `print` function doesn't do any styling. It always renders plain text.

Printing Information

- The `print` function takes a comma-separated list of values
- The values are separated by a space and the last value is followed by a newline
- The values can have different types

```
>>> print(1, 2, 3)
1 2 3
>>> print(1, 'two', 'three', 4.0) # three different types
1 two three 4.0
```

- Calling `print` without arguments produces a newline:

```
>>> print()

>>>
```

You will use the `print` function *a lot*.

Printing Information

- The `print` function has some extra helpful features

```
>>> help(print)
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

- The parameters `sep`, `end`, `file` and `flush` have assignment statements in the header!
- These are called *default parameters*
- Default parameters are values that the parameters take if we don't specify them

We are not concerned with the `file` and `flush` parameters for now.

Printing Information

- We can supply different values by using *keyword arguments* (often called `kwargs`)
- That's just fancy talk for assigning a parameter in a function call:

```
>>> print('a', 'b', 'c') # using the default separator ' '
a b c
>>> print('a', 'b', 'c', sep=', ') # changing the separator to  ', '
a, b, c
```

- Changing the `end` parameter is mostly useful in programs:

```
print('The absolute values of 4 and -5 are ', end=' ')
print(abs(4), ' and ', abs(-5), '.', sep='')
```

```
The absolute values of 4 and -5 are 4 and 5.
```

Similar results can be achieved with escape sequences.

Reading from the Keyboard

- We use the builtin function `input` to read user input from the keyboard
- The `input` function returns what the user enters as a string:

```
>>> species = input()
  Amblyrhynchus cristatus
>>> species
'Amblyrhynchus cristatus'
```

- Even if the input looks like a number:

```
>>> human_population = input() # user enters an estimate as of 26.10.2018 21:57 CEST
  7622432117
>>> human_population
'7622432117'
>>> type(human_population)
<class 'str'>
```

We just accidentally learned about that the `type` function.

Reading from the Keyboard

- If we need a number, we need to use conversion functions:

```
>>> human_population = input() # user enters an estimate as of 26.10.2018 21:57 CEST
  7622432117
>>> human_population = int(human_population)
>>> human_population
7622432117
>>> human_population += 9 # estimate per second as of 26.10.2018 21:57 CEST
>>> human_population
7622432126
```

- We can apply the conversion directly:

```
>>> human_population = int(input()) # user enters an estimate as of 26.10.2018 21:57 CEST
  7622432117
>>> human_population
7622432117
>>> human_population += 9 # estimate per second as of 26.10.2018 21:57 CEST
>>> human_population
7622432126
```

There is a problem if the user doesn't enter a valid integer.

Reading from the Keyboard

- We can tell the user what we want by providing a *prompt*:

```
>>> human_population = int(input('Please enter a population number: '))
Please enter a population number: 7622432117
>>> human_population
7622432117
```

Designing user interaction is hard. Think of the user's perspective, ask for help.

Exercises Lecture 4