15-110 Principles of Computing – F19

Lecture 7:

Strings, Non-Scalar types

Teacher:
Gianni A. Di Caro

So far about Python ...

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators

- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`
  - `input(msg)`

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable

- Conditional flow control

  -
    ```
    if cond_true:
        do_something
    ```

  -
    ```
    if cond_true:
        do_something
    else:
        do_something_else
    ```

  -
    ```
    if cond1_true:
        do_something_1
    elif cond2_true:
        do_something_2
    else:
        do_something_else
    ```

- Data types:
  - `int`
  - `float`
  - `bool`
  - `str`
  - `None`

- Operators:
  - `=`
  - `+`
  - `-`
  - `/`
  - `//`
  - `%`
  - `**`

- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`

- Logical operators
  - `and`
  - `or`
  - `not`

Notation for string literals: single, double, triple quotes

```
'Hi'
'Hello!'
'z'
'abc'
'_wow_'
```

Single quotes

```
"Number 5"
"abc"
"Hello!"
"   "
```

Double quotes

```
"I’m Joe"
'This "trick" is cool!'
```

Double and single quotes together

```
'''This is a very long line of text that it might be convenient
to write over multiple lines to make it well readable.
This is often the case with strings that are used to "describe" a
function or a piece of code'''
```

String operators: + (concatenation)

- **String concatenation**, + operator, *overloaded*: `s = s1 + s2` returns a new string s consisting of the string operands joined together

```
greet_joe = 'Hello Joe'
comma = ','
greet_mary = 'hello Mary'
greet = greet_joe + comma +  greet_mary
print(greet)
```

```
Hello Joe,hello Mary
```

Can I do `greet + 1`? NO!

- **Augmented (*shorthand*) form of** + operator: `s += x`
  - `s` must be already defined
  - Equivalent to `s = s + x`
  - **Works also for numeric types!**

```
s = 'abc'
s1 = ‘defg'
s += s1
```

String operators: * (duplication)

- **String duplication**, * operator, *overloaded:* `sm = n * s` creates a *new* string consisting of **multiple copies** (`n`) of the string `s`
  - `s` is a string and `n` is an integer

```
s = 'Hello'
n = 4
print(s * n)
```

```
HelloHelloHelloHello
```

```
s = 'Hello'
n = 4
s4 = n * s
print(s4)
```

Commutative!

- What if `n` is a negative integer?

```
s = 'Hello'
n = -4
print(s * n)
```

```

```

- **Augmented form of** * operator: `s *= n`
  - `s` must be already defined
  - Equivalent to `s = s * n`
  - **Works also for numeric types!**

- Can I do `s*s`? NO!

String operators: `in` (part of, *membership*)

- **Part of,** `in` operator, *overloaded:* `s in p` returns `True` if the first operand, `s`, is contained within the second, `p`, and `False` otherwise → Membership operator

```
s = 'Joe'
in_hello = s in 'Hello Joe'
in_food = s in 'Yummy meal'
print(in_hello, in_food, type(in_hello))
```

```
True False <class 'bool'>
```

- **Not part of,** `not in` operator, *overloaded:* `s not in p` returns `True` if the first operand, `s`, is *not* contained within the second, `p`, and `False` otherwise

```
s = 'Joe'
in_hello = s not in 'Hello Joe'
in_food = s not in 'Yummy meal'
print(in_hello, in_food, type(in_hello))
```

```
False True <class 'bool'>
```

Scalar vs. Non-scalar objects

- Scalar type objects:
  - `int`
  - `float`
  - `complex`
  - `bool`
  - `None`

Indivisible

- Non-Scalar type literal objects:
  - `str`: **String of ≥ 1 characters (text):**

    `“Hi”, ‘Hello!’, “Number 5”`
  - `tuple`
  - `list`
  - `set`
  - `dict`

- Made of multiple components
- **Individual or subsets of components can be addressed for read and write operations**

Internal structure

➢ Scalar vs. Non-scalar terminology, from *math*

✓ It is termed a scalar any *real number*, or *any quantity* that can be measured using a **single real number**

✓ A vector is made of **multiple scalar components** (represents a point in a multi-dimensional space)

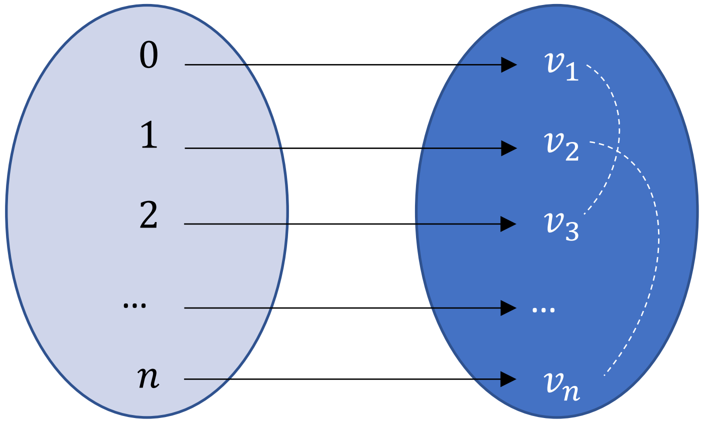
String objects: *Sequences* of characters

- A string is a *sequence* of characters
  - ✓ Sequence → *Ordering*, indexing
  - ✓ Characters → Which type of characters are allowed? → **Unicode set**

- **Sequence:** `'Hello Joe'`

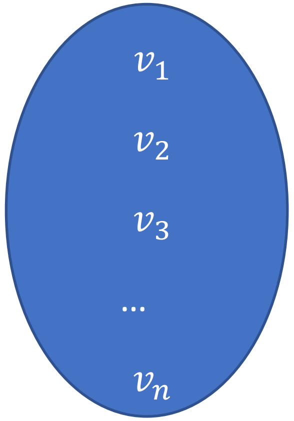
H	e	l	l	o		J	o	e
0	1	2	3	4	5	6	7	8

Indexing of the positions of the individual characters in the string
→ Access to the individual components of the string type



Sequence: duplicate values possible, indexing / order

Set: no duplicates, no indexing / order



String indexing, operator [ ]

- **Indexing:** an integer index is associated to each character to access (read) its value
  - `s[i]` – Operator to **access / read the value of the *i*-th component** of a string variable `s`

String of length `n` = 9 characters

`'Hello Joe'`

H	e	l	l	o		J	o	e
0	1	2	3	4	5	6	7	8

An index `i` must be an integer between 0 and `n-1`

```
greet = 'Hello Joe'
print(greet[0], greet[4], greet[6])
print(greet[9], greet[100]) → Error!
print(greet[-1]) ?
```

String indexing, operator [ ]

- We can also *index from the right end of the string* (useful to get the last character!)

H	e	l	l	o		J	o	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

An index `i` can also be an integer between -1 and -`n`

```
print(greet[-5], greet[-1], greet[4])
print(greet[-9], greet[0])
```

➢ Wrap-up: for a string of `n` characters, a feasible index value to use with `[ ]` operator is any integer between `–n` and `n-1`

Useful built-in functions for strings: Length and Casting

- **`len()`** function*:* Returns the **length of a string** (more in general, of **any non-scalar type**)

```
s = 'Joe and Mary'
length = len(s)
print(s, length)
```

```
Joe and Mary  12
```

- **`str(o)`** function*: Virtually any object* `obj` *in Python can be rendered as a string*. Therefore, `str(obj)` returns the string representation of a python object `obj`:

```
n = 150.5
s = str(n)
print(s)
```

```
150.5
```

String slicing, operator [:]

- **Extracting substrings from a string**, known as string slicing.
- If `s` is a string, the operator `s[m:n]` returns the portion of `s` starting with position `m`, and *up to but not including* position `n`

```
s = 'Hello Joe'
ss = s[0:4]
print(ss)
```

```
Hell
```

- Why isn’t the character at position `n` not included?
  - The result is a string of `m−n` characters

- Slicing can be used to *extract* only the relevant parts of a string

```
dna_strand = 'GTTAGGCCGATTACACATATATA'
start = 8
end = start + 7
interesting = dna_strand[start:end]
```

- Slicing can be used to modify a string, by **creating a new variable** including the desired changes

```
s = 'Hello Joe'
greet = s[0:5] + '! ‘ + s[6:9] + ' and Mary'
```

String slicing, operator [:]

- Shortcuts: omitting indices produces some **default behavior**
  - `s[:4]` is equivalent to `s[0:4]`
  - `s[2:]` is equivalent to `s[2:len(s)]`
  - `s[:]` is equivalent to `s`
    (it’s precisely the same object in memory, same id)

- Default start: 0
- Default end: `len(s)`

- Slicing into empty strings
  - `s[4:2]` returns the empty string
    (start is *positive* and start > end)
  - `s[2:2]` returns the empty string.
    (start and end are the same)

- Slicing with negative indices
  - Extracting from the *right end*
  - `|start| > |end|`
  - `s[0:4]` and `s[-9:-5]` return the same substring

String slicing, operator [:]

- 👉 <u>The *end* value can be equal to the length of the string since the extracted values only go up to `n-1`!</u>
  - `s = 'Hello Joe'`
  - `len(s)` is 9
  - ❖ → `s[9]` generates error, indices go from 0 to 8
  - ✓ `x = s[6:9]` works, with `x ='Joe'` since the value at the 9-th position is not extracted in the slice

String slicing with a *stride:* operator [::]

- **Extracting substrings of non (necessarily) adjacent characters from a string**
- Known as slicing with a stride
- If `s` is a string, an expression of the form `s[m:n:s]` returns the portion of `s` starting with position `m`, and up to but not including position `n`, with the third index `s` designating a *stride* (a step length), which indicates how many characters to *jump* after retrieving each next character in the slice

```
s = 'Hello Joe'
ss = s[0:9:2]
print(ss)
```

```
HloJe
```

```
alphabet = 'abcdefghijklmanoprsqtuvwxyz'
even_letters = alphabet[::2]
odd_letters = alphabet[1::2]

second_half = alphabet[len(alphabet)//2:]
first_third = alphabet[:len(alphabet)//3]
every_three_letters_sh = second_half[::3]

print("Is 'w' an even letter?", 'w' in even_letters)
print("Is 'q' in the second half?", 'q' in second_half)
```

String slicing with a *stride:* operator [::]

- Indices can be omitted: first and second indices default to the first and last characters respectively, while the third defaults to 1
  - `s = 'Hello Joe'`
  - `s[::4]` is equivalent to `s[0:9:4]`
  - `s[:6:2]` is equivalent to `s[0:6:2]`
  - `s[1:6:]` is equivalent to `s[1:6:1]`
  - `s[::]` is equivalent to `s` (it's the same object)

- Default start: 0
- Default end: `len(s)`
- Default step: 1

- Striding with negative steps
  - Extracting from the *right end*
  - Steps are negative
  - `|start| > |end|`

- `s[4:0:-1]` gives 'olle'
- `s[::-1]` gives 'eoJ olleH'