Programming in Python

Data structures in Python
lecture 3

Department of Cybernetics and Artificial Intelligence
Technical University of Kosice
Ing. Ján Magyar, PhD.

Strings in Python

- immutable sequence of characters
- defined within quotation marks or apostrophes

```
"Hello World!"          'Hello World!'
```

- you can define strings on multiple lines with threefold quotes and apostrophes (whitespaces will be a part of the string)

```
"""Hello                '''Hello
   World!"""               World!'''
```

- you can create strings by changing the type of a value, e.g.

```
str(5)
```

Encoding strings

- in Python 3 usually not necessary
- converting strings to bytes and vice versa
  - `encode()` - string → bytes
  - `decode()` - bytes → string

```
nonlat = '字'
b = nonlat.encode()          b'\xe5\xad\x97'
b.decode()                   '字'
```

Modifying strings

- adding a string to the end of an existing string

```
test = "Hello"
test += " World!"
```

- concatenating strings

```
test = "Hello" + " World!"
test = "Hello" " World!"
```

- using the `join` function

```
test = ' '.join(["Hello", "World!"])
```

Strings as lists

- Python considers strings to be a special type of lists
- we can access individual characters using indexes

```
test = "Hello World!"
test[0] -> 'H'
```

- Python supports negative indexing (from the end)

```
test[-1] -> '!'
```

Strings as lists - slicing

- to get a part of a string, use slicing

```
test[2:7] -> 'llo W'
test[:7] -> 'Hello W'
test[2:] -> 'llo World!'
test[2:-3] -> 'llo Wor'
test[:] -> 'Hello World!'
```

- the third parameter represents step (every *i*th character)

```
test[2:7:2] -> 'loW'
```

Finding the length of a string/sequence

- `len(string)`
- an empty string has a length of `0`
- valid indexes in a string: from `0` to `len(string) - 1`

Selected string methods - search

- `find(sub[, start[, end]])` / `rfind(sub[, start[, end]])`
  - finds the first occurrence of a substring (`sub`) in `string[start:end]`
  - returns `-1` if `sub` does not occur in the string
- `index(sub[, start[, end]])` / `rindex(sub[, start[, end]])`
  - finds the first occurrence of a substring (`sub`) in `string[start:end]`
  - interpreter throws a `ValueError` if `sub` does not occur in the string
- `count(sub[, start[, end]])`
  - returns the number of occurrences of `sub` in `string[start:end]`
  - overlapping occurrences do not count

Selected string methods - splitting

- `split([sep[, maxsplit]]) / rsplit([sep[, maxsplit]])`
  - splits a string into individual parts along a separator (`sep`)
  - `maxsplit` defines the maximum number of splits (number of parts: `maxsplit + 1`)
  - `sep` is whitespace by default
- `partition(sep) / rpartition(sep)`
  - splits a string into three parts along the first occurrence of a separator (`sep`)
  - returns a triple: substring before the separator, separator, substring after the separator
  - if the separator does not occur in the string, it returns a triple with the whole string and two empty strings
- `splitlines([keepends])`
  - splits a string into lines
  - uses multiple characters as separators that can represent a newline, not only `\n`
  - if `keepends` is `True`, the resulting strings contain the newline character

Selected string methods - update

- `replace(old, new[, count])`
  - returns a copy of the string with occurrences of `old` replaced with `new`
  - `count` limits the number of replaced occurrences
- `maketrans(x[, y[, z]])`
  - creates a table for updating multiple characters in the string for `translate`
  - if we enter only `x`, it must be a dictionary mapping old characters to new ones
  - if we pass `x` and `y`, they must be strings with the same length defining mapping
  - `z` is a string of characters that will be removed from the copy
- `translate(table)`
  - returns a copy of the string with replaced characters

Selected string methods - formatting first letters

- `capitalize()`
  - returns a copy of the string with capital first letter; other letters are small
- `title()`
  - returns a copy of the string in which each word starts with a capital letter
  - cannot process apostrophes etc., any group of letters is considered a word
- `upper()`
  - returns a copy of the string with only uppercase letters*
- `lower()`
  - returns a copy of the string with only lowercase letters *
- `casefold()`
  - returns a copy of the string with only lowercase letters
  - used for case-insensitive comparisons

Selected string methods - checking

- `endswith/startswith(sub[, start[, end]])`
  - returns `True` if `string[start:end]` ends or starts with `sub`
  - `sub` can be a tuple of multiple potential substrings
  - `start` and `end` are optional arguments
- `islower() / isupper()`
  - returns `True` if all letters are lowercase or uppercase in the string
  - if the string contains no letters, it returns `False`
- `istitle()`
  - returns `True` if each word starts with a capital letter and other letters are lowercase
  - if the string contains no letters, it returns `False`

Selected string methods - adjust

- `lstrip([chars]) / rstrip([chars]) / strip([chars])`
  - returns a copy of the string with character in `chars` removed from the start/end
- `ljust(width[, fillchar]) / rjust(width[, fillchar]) / center(width[, fillchar])`
  - returns a string with a length of `width` containing the original string adjusted to the left/right/centre
  - `fillchar` is the character used to fill the space – whitespace by default
  - if `width <= len(string)`, the method returns the original string
- `zfill(width)`
  - returns a copy of the string with a length of `width` representing a number with leading zeros added to the beginning (with sign added if needed)
  - if `width <= len(string)`, the method returns the original string

Selected string methods - control

- `isalpha()`
  - returns `True` if every character is a letter from the unicode database, returns `False` for empty strings
- `isdecimal()`
  - returns `True` if every character can be used for representing a decimal number, returns `False` for empty strings
- `isdigit()`
  - returns `True` if every character is a digit, returns `False` for empty strings
- `isnumeric()`
  - returns `True` if every character is a number, returns `False` for empty strings
- `isalnum()`
  - returns `True` if every character meets at least one condition

String formatting

- `format()`
- placeholder is denoted with `{}`
- placeholder implicitly refers to the arguments and keeps their order
  - `"Test {}".format("string")`
  - `"{} {}".format("Test", "string")`
  - `"Test {0}".format("string")`
  - `"Hey {name}".format(name="Jude")`
  - `"Object name: {0.name}".format(object)`
  - `"List head: {lst[0]}".format(lst=mylist)`

String preprocessing

- when calling `format()` you can preprocess strings using functions
  - `str()` - changing the type to string
    ```
    "Make it a string: {!s}".format("test")
    ```
  - `repr()` - changing the type to string, adding quotation marks
    ```
    "Add quotes: {!r}".format("test")
    ```
  - `ascii()` - changing the type to ASCII string, adding quotation marks
    ```
    "Make it ASCII: {!a}".format("test")
    ```

String adjust in formatting

- we can adjust strings to a set length
  - `"{:<30}".format('left aligned')`
  - `"{:>30}".format('right aligned')`
  - `"{:^30}".format('center aligned')`
  - `"{:-^30}".format('center with fill char')`

Printing signs

- `"Show always: {:+f}; {:+f}".format(19.09, -19.09)`
- `"Show space for positives: {: f}; {: f}".format(19.09, -19.09)`
- `"Show only minus: {:-f}; {:-f}".format(19.09, -19.09)`
- `"Show only minus: {:f}; {:f}".format(19.09, -19.09)`

Formatting numbers

```
"int: {0:d}, hex: {0:x}, oct: {0:o}, bin:
{0:b}".format(42)

"Add separator: {:,}".format(21081968)

"Round float numbers:
{0:.2f}".format(19.9499999999999)

"Percentage: {:.2%}".format(true/total)
```

Formatting with dictionaries

```
person_dict = {
     "name": "Roman",
     "age": 32
}
"{name} is {age} years old".format(**person_dict)
```

Lists in Python

- an ordered set of values
- members are mutable
- can contain duplicate values
- can contain values of different types (not typical)
- data structure, not primitive type

Working with lists

- creating a list

```
lst = [1, 2, 3]
lst = []
lst = list()
```

- indexing and slicing – as with strings
- updating a value

```
lst[0] = "a"
lst[1:3] = ["b", "c"]
```

Working with lists

- iterating over a list

```
for element in lst:
    print(element)
for idx, element in enumerate(lst):
    print(idx, element)
```

- search

```
element [not] in lst
```

- finding the index of the first occurrence (`ValueError` if no occurrence is found)

```
lst.index(elem[, start[, end]])
```

- finding the length

```
len(lst)
```

- flipping the order

```
lst.reverse()
```

Adding values to the list

- adding to the end
  ```
  lst.append(4)
  ```
- adding to a certain index
  ```
  lst.insert(2, "new element")
  ```
- list concatenation
  ```
  lst = lst + [6, 7, 8]
  lst.extend([6, 7, 8])
  ```

Removing members from the list

- removing a certain element (`ValueError` if not found)
  ```
  lst.remove('a')
  ```
- removing from position
  ```
  lst.pop(2)
  del lst[2]
  ```
- removing the list contents
  ```
  lst.clear()
  del lst[:]
  ```
- removing the list
  ```
  del lst
  ```

Ordering the list

- members must be of the same type (or at least comparable)
- `lst.sort(key=None, reverse=False)`
  - ascending by default (from smallest to largest)
  - for descending order - `reverse=True`
  - you can define the key used for sorting, usually with lambda expressions

Copying – primitive types vs. data structures

```
x = 5
y = x
x = 6
```

What's the value of `x` and `y`?

```
lst1 = ['a', 'b', 'c']
lst2 = lst1
lst1.append('d')
```

What's the value of `lst1` and `lst2`?

```
lst1 = ['a', 'b', 'c']
lst2 = lst1
lst1 = [4, 5]
```

What's the value of `lst1` and `lst2`?

Copying lists

- `lst.copy()`
  - returns a shallow copy (members are not copied)
- `deepcopy()`
  - returns a deep copy
  - for lists of lists
  - `from copy import deepcopy`

List comprehensions

- for creating lists which contain values following a certain
- general syntax

```
lst = [element_tba for element in enumeration]
```

- cube of the first 10 numbers

```
lst = []
for num in range(1, 11):
    lst.append(num ** 3)

lst = [num ** 3 for num in range(1, 11)]
```

List comprehensions with conditions

- even numbers up to 100

```
lst = [num for num in range(101) if num % 2 == 0]
```

- vowels from a word (changing to uppercase letters)

```
lst = [str.upper(c) for c in word
       if lower(c) in ['a', 'e', 'i', 'o', 'u']]
```

- list of files from a directory

```
from os import listdir
from os.path import join, isfile
files = [join(dir, x) for x in listdir(dir)
         if isfile(join(dir, x))]
```

Nested list comprehensions

- all possible combinations of values from two lists

```
combinations = [(x, y) for x in [1, 2] for y in [3, 1]]
```

- transposing a matrix

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]
i_matrix = [[row[i] for row in matrix]
            for i in range(len(matrix[0]))]
```

Tuples

- ordered set of values
- members are immutable
- can contain duplicates
- usually values of different types
- data structure, not a primitive type

Creating a tuple

- listing values
```
sample_tuple = (1, 2, 3)
```
- creating a tuple with a single value
```
sample_tuple = (1, )
sample_tuple = 1,
```
- creating an empty tuple
```
sample_tuple = tuple()
sample_tuple = ()
```

Tuple operations (as with lists)

```
x in tuple
x not in tuple
tuple1 + tuple2
tuple * num
tuple[i]
tuple[start:end:step]
len(tuple)
min(tuple)/max(tuple)
tuple.index(element[, start[, end]])
tuple.count(element)
```

Dictionary

- data structure with key-value pairs
- members are not ordered
- uses general indexing instead of numbers
- typical use cases
  - dictionaries (multi-language support)
  - changing representations (e.g. 1 → one, 2 → two)
  - hash table

Creating a dictionary

- empty dictionary

```
my_dict = {}
my_dict = dict()
```

- listing members

```
my_dict = {
        1: "one",
        2: "two"
}
```

- adding a member

```
my_dict[3] = "three"
```

Accessing dictionary members

- `key [not] in dictionary`
  - determines whether a key is (not) present in a dictionary
- `my_dict[key]`
  - `KeyError` if the key is not present
- `my_dict.get(key[, default])`
  - returns `mydict[key]` if it exists
  - if the key was not found, returns `None`
  - if `default` was set and the key is not present in the dictionary, it returns `default`

Traversing a dictionary

- `for x in dictionary`
  - for loop for keys in dictionary
- `dictionary.keys()`
  - returns a view of the dictionary's keys, we can make it into a list `list(dictionary.keys())`
- `dictionary.values()`
  - returns a view of the dictionary's values, we can make it into a list `list(dictionary.values())`
- `dictionary.items()`
  - returns a view of the dictionary's key-value pairs, we can make it into a list `list(dictionary.items())`

Removing from a dictionary

- `del dictionary[key]`
  - removes the key-value pair from the dictionary with key `key`; if it does not exist, throws `KeyError`
- `dictionary.pop(key[, default])`
  - removes and returns the value of the key-value pair from the dictionary with `key`
  - if it does not exist, returns `default` (if `default` was not set, throws `KeyError`)
- `dictionary.popitem()`
  - removes and returns the key-value pair added as last to the dictionary
- `dictionary.clear()`
  - removes the dictionary's contents
- `del dictionary`
  - deletes the dictionary

Further dictionary functions

- `dictionary.copy() / deepcopy()`
  - copying as with lists
- `dictionary.setdefault(key[, default])`
  - adds a key-value pair `dictionary[key] = default` if the key is not yet present, returns `default`
  - if the key is already present, it returns the corresponding value
  - `default` is `None` by default
- `dictionary.update([other])`
  - updates dictionary with key-value pairs in `other`
  - `other` can be
    - dictionary
    - set of tuples or other sequences of length 2
    - can pass pairs as arguments

```
dictionary.update(one=1)
```

Conclusion

- working with strings
- working with lists
- working with tuples
- working with dictionaries
- difference between assigning and copying