

Lecture 4

Interactive programs and manipulation of data

Lecture 4 - Interactive programs

- Interact with your program
 - function `input()`: the program prints a first prompt (request) and **waits for the user** to enter some string via terminal
 - write your text in the terminal
 - press ENTER to send text
 - returns always a string, containing the user input
 - store the result: `user_input = input("request text")`
 - then, the program continues with the next instructions

```
age = input("insert your age here: ")  
print("Your age is: " + age)
```



```
[marco@mr-katana] ~  
• > /usr/bin/python "/home/marco/Universita/Corsi  
insert your age here: 57  
Your age is: 57
```

Lecture 4 - Interactive programs

- Interact with your program
 - input text has always type <string>
 - for example, we cannot do:

```
age = input("insert your age here: ")
print("Your age is: " + age)
prev_age = age - 1
```

→

```
insert your age here: 34
Your age is: 34
Traceback (most recent call last):
  File "/home/marco/Universita/Corsi e lezioni/ISEG/Prog-T-Python", line 3, in <module>
    prev_age = age - 1
                ~~~~^
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

- We need to convert the string into some other type (*type casting*)
 - we already saw the cast integer to string with `str()`
 - the reverse is similar: use the appropriate “type function” such as: `int()`, `float()`...

```
age = input("insert your age here: ")
print("Your age is: " + age)
prev_age = int(age) - 1
print("Your age last year was: " + str(prev_age))
```

→

```
[marco@mr-katana] ~
> /usr/bin/python "/home/marco/Universita/Corsi e lezioni/ISEG/Prog-T-Python"
insert your age here: 34
Your age is: 34
Your age last year was: 33
```

Lecture 4 - Type casting

- Be careful: you have to use the correct type conversion functions
 - Numeric strings can be converted to numbers
 - "42" → int ✓ "3.14" → float ✓
 - The string must be compatible with the target type
 - "hello" → int ✗ "3.14" → int ✗
 - float → int removes the decimal part (it does not round)
 - int(3.9) → 3
 - Numbers can always be converted to strings
 - str(42) → "42"
 - Boolean conversion follows truthiness rules
 - 0, 0.0, "" → False
 - non-zero numbers and non-empty strings → True
 - therefore bool("False") → True
- An invalid numeric conversion typically raises a ValueError.

```
# Type conversion

int("42")      # 42      str → int
float("3.14")  # 3.14    str → float
str(42)        # "42"    int → str
bool(1)        # True    int → bool

int(3.9)       # 3       float → int
float(5)       # 5.0     int → float

# Invalid conversions

int("hello")    # ValueError
int("3.14")     # ValueError
float("hello")  # ValueError

# Valid but perhaps unexpected

bool("False")   # True
bool("0")       # True
bool("")        # False
```

Lecture 4 - String manipulation

- Strings are not a whole block: they are a **sequence of chars**
 - It is possible to retrieve the number of chars (length of the string) with `len()` function
 - Each char has an **index**
 - the first is 0, the last is `len(s)-1` (otherwise: **IndexError: string index out of range.**)

chars	H	a	r	r	y		P	o	t	t	e	r
Index	0	1	2	3	4	5	6	7	8	9	10	11

- retrieve the char from its index `i` with `[]`

```
s = 'Python'
s_len = len(s)      # = 6
first = s[0]        # = 'P'
last = s[s_len-1]   # = 'y'

#ERROR:
s[1] = 'i'
# TypeError: 'str' object
# does not support item assignment
```

Lecture 4 - String manipulation

- Strings are not a whole block: they are a **sequence of chars**
 - It is possible to retrieve the number of chars (length of the string) with `len()` function
 - Each char has an **index**
 - the first is 0, the last is `len(s)-1` (otherwise: **IndexError: string index out of range.**)

chars	H	a	r	r	y		P	o	t	t	e	r
Index	0	1	2	3	4	5	6	7	8	9	10	11

- retrieve the char from its index `i` with `[]`

In Python, strings are **immutable** (we will see).
This means we cannot modify it.
We can still assign the same variable `s` to a new string, or constructing new “updated” ones.

```
s = 'Python'
s_len = len(s)      # = 6
first = s[0]        # = 'P'
last = s[s_len-1]   # = 'y'

#ERROR:
s[1] = 'i'
# TypeError: 'str' object
# does not support item assignment
```

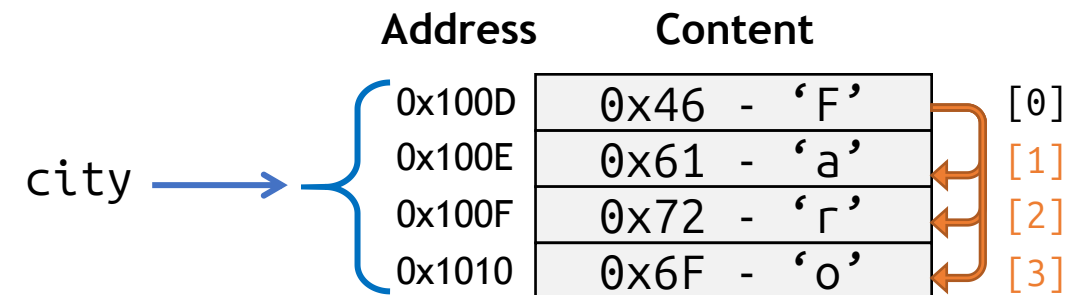
Lecture 4 - String manipulation

- Negative index:
 - It is possible to get values counting backward, from the last element
 - Python uses negative indexes, where
 - `string[-1]` → last char (= `string[ len(string) - 1 ]`)
 - `string[-2]` → second-to-last (= `string[ len(string) - 2 ]`)
 - ...
 - `string[ -len(string) ]` → first (= `string[ 0 ]`)
 - Negative indices allow us to access a character at a known offset from the end without computing the string length.

chars	H	a	r	r	y		P	o	t	t	e	r
Index	0	1	2	3	4	5	6	7	8	9	10	11
Index	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Lecture 4 - String manipulation

- Relation between **index** and **memory**
 - Each character is identified by its position (index) in the sequence.
 - At a low level, data is stored in memory locations identified by addresses.
 - The **index** allows Python to locate the corresponding element, while hiding the details of its physical memory representation.



Variable → reference to the string object in memory
Index → position of a character within the string

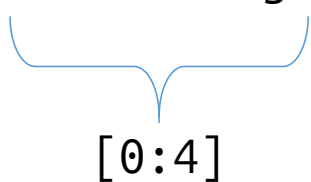
Lecture 4 - String slicing

- Slicing allows us to extract a portion of a string
 - General syntax: `string[start:stop]`
 - start → index of the **first character included**
 - stop → index of the **first character excluded**
 - If start is omitted → start from the beginning
 - If stop is omitted → continue to the end

```
text = "Programming"
```

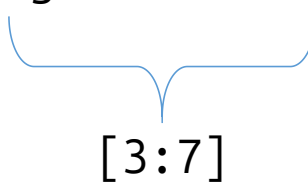
```
text[0:4]    # "Prog"
text[3:7]    # "gram"
text[:4]     # "Prog"
text[4:]     # "ramming"
text[:]      # "Programming"
```

0	1	2	3	4	5	6	7	8	9	10
P	r	o	g	r	a	m	m	i	n	g



[0:4]

0	1	2	3	4	5	6	7	8	9	10
P	r	o	g	r	a	m	m	i	n	g



[3:7]

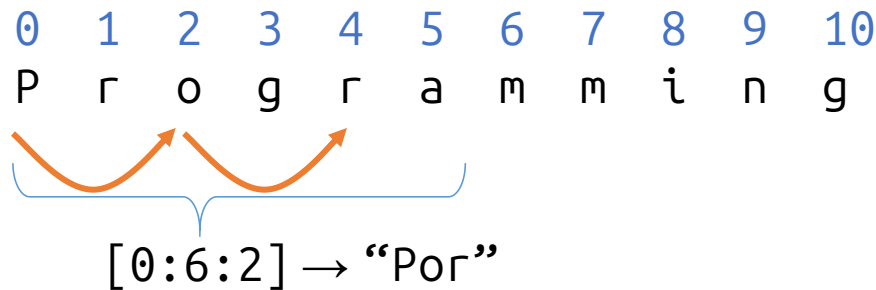
Lecture 4 - String slicing

- Slicing with a step
 - General syntax: `string[start:stop:step]`
 - step determines how many positions to move
 - default = 1
 - a negative step moves backwards
 - With a negative step, slicing moves from right to left. Therefore, start must normally be greater than stop.

```
text[2:]      # "llo"
text[: -1]    # "Hell"
text[::2]     # "Hlo"
text[:: -1]   # "olleH"

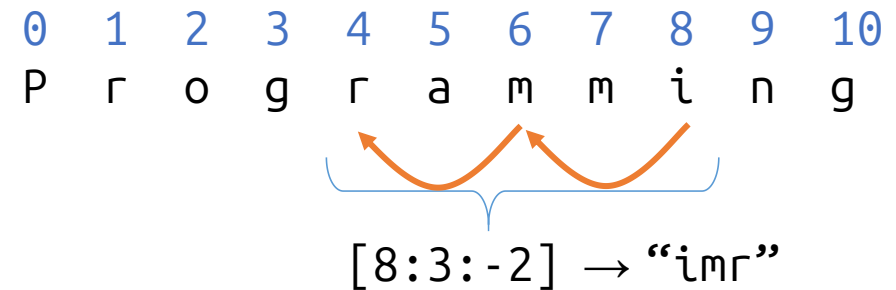
text[:: -1]   # reverse the whole string
text[6:2: -1] # reverse a portion
text[6:: -1]  # from index 6 back to the beginning
```

0 1 2 3 4 5 6 7 8 9 10
P r o g r a m m i n g



[0:6:2] → "Por"

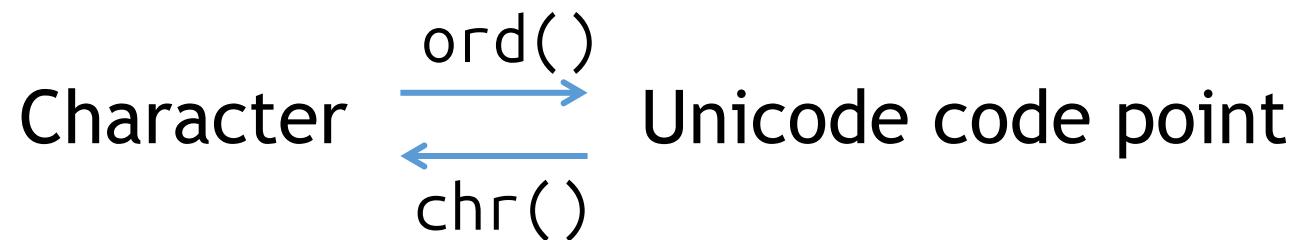
0 1 2 3 4 5 6 7 8 9 10
P r o g r a m m i n g



[8:3:-2] → "imr"

Lecture 4 - Strings and Unicode

- Python strings support Unicode characters directly.
- two built-in functions to convert between a character and its Unicode code point:



```
ord("A")      # 65      → U+0041
ord("ã")      # 227     → U+00E3
ord("€")      # 8364    → U+20AC
ord("😊")      # 128512  → U+1F600
```

```
chr(65)       # 'A'
chr(227)      # 'ã'
chr(8364)     # '€'
chr(128512)   # '😊'
```

```
text = "João"
currency = "€"
emoji = "😊"

print(text)      # João
print(currency)  # €
print(emoji)     # 😊
```

Lecture 4 - String and Unicode

- If strings are indicated between “” or ‘’... How do we write quotes?
- **Escape Sequences:**
 - Allow us to represent special characters inside a string.
 - They start with a backslash \
- Unicode characters can also be represented using escape sequences:

```
print("\u00E3")    # ã
print("\u20AC")    # €
print("\U0001F600") # 😊
```

```
print("Hello\nWorld")    # \n → new line
print("Hello\tWorld")    # \t → tab
print("He said \"Hello\"") # \" → double quote
print('It\'s Python')    # \' → single quote
print("C:\\Users\\Marco") # \\ → backslash
```

Further Reading and exercises:

<https://www.programiz.com/python-programming/string>

Full table: https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences

Lecture 4 - Functions VS Methods

- Python is Object-Oriented → any value is an **object** (specific instance of a type)
- an object, instead of a simple value, may have different *attributes* and *methods*, depending on its type.
- Methods are similar to functions since they perform operations, but they are called differently
 - functions called by their name, data passed as argument: `function(arguments)`
 - methods are called directly on the object: `object.method(arguments)`
 - The dot means: “apply this method to this object.”
 - Both may return a value that can be stored in a variable

```
len(text)           → function  
text.upper()        → method
```

Lecture 4 - Useful string methods

- Python strings provide many methods for creating modified versions of a string:
 - `.lower()` → substitute all letters with the lower case
 - `.upper()` → substitute all letters with the upper case
 - `.replace()` → substitute a specific patten with another given as an argument
 - NOTE: by default, it replaces **every** match found, but it can be set a limit of occurrences

```
text = "Hello Python!"

lower = text.lower() # "hello python!"
upper = text.upper() # "HELLO PYTHON!"

new_text = text.replace("Python", "Bob")
# "Hello Bob!"
```

full list of string methods and examples: <https://www.programiz.com/python-programming/methods/string>

Lecture 4 - Useful string methods

- Python strings provide many methods for creating modified versions of a string:

```
text = "Hello Python!"

lower = text.lower() # "hello python!"
upper = text.upper() # "HELLO PYTHON!"

new_text = text.replace("Python", "Bob")
# "Hello Bob!"
```

Remember that strings are *immutable*: String methods do not modify the original string. They return a new string.

```
text = "Hello Python!"
text.upper()
print(text)      # "Hello Python!"
```

Lecture 4 - Formatted Strings

- f-strings allow us to insert variable values directly inside a string
- Syntax: `f"text {expression} text"`
- **Number formatting**
 - example: `:.2f` displays a floating-point value with 2 decimal places

```
total = 37.5

print(f"Total: €{total:.2f}")
#Total: €37.50
```

```
name = "Sofia"
age = 24

print(f"My name is {name} and I am {age} years old.")
# My name is Sofia and I am 24 years old.

price = 12.5
quantity = 3

print(f"Total: €{price * quantity}")
# Total: €37.5
```

f-strings make formatted output more readable and easier to write

Lecture 4 - Formatted Strings

- f-strings allow us to insert variable values directly inside a string
- Syntax: `f"text {expression} text"`
- **Number formatting**
 - example: `:.2f` displays a floating-point value with 2 decimal places

```
total = 37.5

print(f"Total: €{total:.2f}")
#Total: €37.50
```

```
name = "Sofia"
age = 24

print(f"My name is {name} and I am {age} years old.")
# My name is Sofia and I am 24 years old.

price = 12.5
quantity = 3

print(f"Total: €{price * quantity}")
# Total: €37.5
```

f-strings make formatted output more readable and easier to write

Hands-on Lab!