

Corso Python

Lecture A02 – Structured data types

Alberto Montresor

Università di Trento

2021/01/30

Acknowledgments: Stefano Teso

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



Table of contents

- 1 Strings
- 2 Lists
- 3 Tuples
- 4 Dictionaries

Strings

Strings

Strings are **immutable** objects containing text, represented as a sequence of characters.

- Strings are immutable: they can be read, but all operations that appear to modify them actually create a new string.
- Strings are a sequential collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right.
- A string that contains no characters, often referred to as the **empty string**, is still considered to be a string.

How to define strings

```
>>> print('I am a single quoted string')  
I am a single quoted string
```

```
>>> print("I am a double quoted string")  
I am a double quoted string
```

```
>>> print("""I am a triple quoted string""")  
I am a triple quoted string
```

```
>>> print("")
```

Escaped characters

Some characters cannot be represented directly, so they need to be escaped, i.e. prefixed with \ (backslash)

```
>>> print("So I said, \"You don't know me!\")  
So I said, "You don't know me!"
```

```
>>> print('So I said, "You don\'t know me!"')  
So I said, "You don't know me!"
```

```
>>> print("This will print only three backslashes: \\ \\ \\")  
This will print only three backslashes: \ \ \
```

```
>>> print("""The double quotation mark (\") is used to...""")  
The double quotation mark (") is used to...
```

Escaped characters

\\	Backslash
\n	ASCII linefeed (also known as newline)
\t	ASCII tab character
\'	Single quote
\"	Double quote
\xxxx	Unicode character xxxx (hexadecimal)

```
sad_joke = "Time flies like an arrow.\nFruit flies like a banana."  
print(sad_joke)
```

```
sad_joke = """Time flies like an arrow.  
Fruit flies like a banana."""  
print(sad_joke)
```

String-number conversion

Built-in functions

<code>str(n)</code>	convert number <code>n</code> into a string
<code>int(s)</code>	convert string <code>s</code> into an integer
<code>float(s)</code>	convert string <code>s</code> into a float

```
n = 10
s = str(n)
print(n, type(n))
print(s, type(s))
```

```
10 <class 'int'>
10 <class 'str'>
```

```
n = int("123")
f = float("1.23")
print(n, type(n))
print(f, type(f))
```

```
123 <class 'int'>
1.23 <class 'float'>
```

String-number conversion

```
>>> print(int("3.14"))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: invalid literal for int() with base 10: '3.14'
```

```
>>> print(float("one"))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: could not convert string to float: 'one'
```

```
>>> print(int("1,000"))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: invalid literal for int() with base 10: '1,000'
```


String operators

Result	Operator	Meaning
int	<code>len(str)</code>	Return the length of the string
str	<code>str + str</code>	Concatenate two strings
str	<code>str * int</code>	Replicate the string
bool	<code>str in str</code>	Check if a string is present in another string
str	<code>str[int]</code>	Read the character at specified index
str	<code>str[int:int]</code>	Extract a sub-string

Concatenation

```
s1 = "one" + " " + "string"
length = len(s1)
print("the string:", s1, "is", length, "characters long")

s2 = "hello,"*3
print("the string: ", s2, "is", len(s2), "characters long")
```

Concatenation

```
s1 = "one" + " " + "string"
length = len(s1)
print("the string:", s1, "is", length, "characters long")

s2 = "hello,"*3
print("the string: ", s2, "is", len(s2), "characters long")
```

```
the string: one string is 10 characters long
the string: hello,hello,hello, is 18 characters long
```

Warning: Concatenation with integers

Python

```
>>> var = 123
>>> print("The value of var is " + var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

>>> print("The value of var is " + str(var))
The value of var is 123
```

Java

```
System.out.println("The value of var is " + var)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)
print("beautiful" in s)
print("BEAUTIFUL" in s)
print("ul jour" in s)
print("Gengis Khan" in s)
print(" " in s)
print("  " in s)
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)
print("BEAUTIFUL" in s)
print("ul jour" in s)
print("Gengis Khan" in s)
print(" " in s)
print("  " in s)
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
```

```
print("beautiful" in s)  True
```

```
print("BEAUTIFUL" in s)
```

```
print("ul jour" in s)
```

```
print("Gengis Khan" in s)
```

```
print(" " in s)
```

```
print("  " in s)
```

```
print(s in s)
```

```
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)
print("Gengis Khan" in s)
print(" " in s)
print("  " in s)
print(s in s)
print("" in s)
```


in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s)
print(" " in s)
print("  " in s)
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s) False
print(" " in s)
print("  " in s)
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s) False
print(" " in s)          True
print("  " in s)
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"
```

```
print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s) False
print(" " in s)          True
print("  " in s)         False
print(s in s)
print("" in s)
```

in operator

```
s = "A beautiful journey"

print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s) False
print(" " in s)          True
print("  " in s)         False
print(s in s)            True
print("" in s)           True
```

in operator

```
s = "A beautiful journey"

print("A" in s)           True
print("beautiful" in s)  True
print("BEAUTIFUL" in s)  False
print("ul jour" in s)    True
print("Gengis Khan" in s) False
print(" " in s)          True
print("  " in s)         False
print(s in s)            True
print("" in s)           True
```

String Indexing

Character extraction

You can extract a character located at index `i` of string `s` with the expression `s[i]`

String extraction (**slicing**)

You can extract a substring of a string `s` with the expressions:

<code>s[start:end]</code>	Returns the characters located between index <code>start</code> (included) and index <code>end</code> (excluded)
<code>s[:end]</code> (prefix)	Returns the characters located between the beginning of the string and index <code>end</code> (excluded)
<code>s[start:]</code> (suffix)	Returns the characters located between index <code>start</code> (included) and the end of the string

Single characters

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
print(s[0], s[2], s[len(s)-1])
print(s[-1], s[-3], s[-5])
print(s[len(s)])
```

Single characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Luther College"
print(s[0], s[2], s[len(s)-1])
print(s[-1], s[-3], s[-5])
print(s[len(s)])
```

L t e

Single characters

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
print(s[0], s[2], s[len(s)-1])
print(s[-1], s[-3], s[-5])
print(s[len(s)])
```

```
L t e
e e l
```

Single characters

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Luther College"
print(s[0], s[2], s[len(s)-1])
print(s[-1], s[-3], s[-5])
print(s[len(s)])
```

```
L t e
e e l
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])
```

```
print(s[0:2])
```

```
print(s[0:5])
```

```
print(s[:5])
```

```
print(s[-5:-1])
```

```
print(s[-5:])
```

```
print(s[3:-3])
```

```
print(s[:])
```

Extraction (Slicing)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Luther College"
```

```
print(s[0:1])
```

```
print(s[0:2])
```

```
print(s[0:5])
```

```
print(s[:5])
```

```
print(s[-5:-1])
```

```
print(s[-5:])
```

```
print(s[3:-3])
```

```
print(s[:])
```

Extraction (Slicing)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])
print(s[0:5])
print(s[:5])
print(s[-5:-1])
print(s[-5:])
print(s[3:-3])
print(s[:])
```

Extraction (Slicing)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])
print(s[:5])
print(s[-5:-1])
print(s[-5:])
print(s[3:-3])
print(s[:])
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])           Luthe
print(s[:5])            Luthe
print(s[-5:-1])        Luthe
print(s[-5:])           Luthe
print(s[3:-3])         llege
print(s[:])             Luther College
```


Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])           Luthe
print(s[:5])            Luthe
print(s[-5:-1])         Luthe
print(s[-5:])           Luthe
print(s[3:-3])          llege
print(s[:])             Luther College
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])           Luthe
print(s[:5])            Luthe
print(s[-5:-1])         lleg
print(s[-5:])           lleg
print(s[3:-3])          l
print(s[:])             Luther College
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])           Luthe
print(s[:5])            Luthe
print(s[-5:-1])        lleg
print(s[-5:])           llege
print(s[3:-3])         llege
print(s[:])             Luther College
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])           Lu
print(s[0:5])           Luthe
print(s[:5])            Luthe
print(s[-5:-1])        lleg
print(s[-5:])           llege
print(s[3:-3])         her Coll
print(s[:])            Luther College
```

Extraction (Slicing)

0 1 2 3 4 5 6 7 8 9 10 11 12 13

L	u	t	h	e	r		C	o	l	l	e	g	e
---	---	---	---	---	---	--	---	---	---	---	---	---	---

-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

```
s = "Luther College"
```

```
print(s[0:1])           L
print(s[0:2])          Lu
print(s[0:5])          Luthe
print(s[:5])           Luthe
print(s[-5:-1])       lleg
print(s[-5:])          llege
print(s[3:-3])        her Coll
print(s[:])            Luther College
```

Methods

Result	Method	Meaning
str	<code>str.upper()</code>	Return the string in upper case
str	<code>str.lower()</code>	Return the string in lower case
str	<code>str.strip(str)</code>	Remove strings from the sides
str	<code>str.lstrip(str)</code>	Remove strings from the left
str	<code>str.rstrip(str)</code>	Remove strings from the right
str	<code>str.replace(str, str)</code>	Replace substrings
bool	<code>str.startswith(str)</code>	Check if the string starts with another
bool	<code>str.endswith(str)</code>	Check if the string ends with another
int	<code>str.find(str)</code>	Return the first position of a substring starting from the left
int	<code>str.rfind(str)</code>	Return the position of a substring starting from the right
int	<code>str.count(str)</code>	Count the number of occurrences of a substring

Stripping and replacing

Stripping removes the specified characters from the beginning or the end of the string. If not specified, removes spaces

```
text = "   one piece   "
print("|" + text.strip() + "|")
print("|" + text.lstrip() + "|")
print("|" + text.rstrip() + "|")

text = "xoxo -one piece- xoox"
print("|" + text.strip(" xo") + "|")
print("|" + text.lstrip(" xo") + "|")
print("|" + text.rstrip(" xo") + "|")

print(text.replace("xo", "*"))
```

Stripping and replacing

Stripping removes the specified characters from the beginning or the end of the string. If not specified, removes spaces

```

text = "   one piece   "
print("|" + text.strip() + "|")           |one piece|
print("|" + text.lstrip() + "|")         |one piece  |
print("|" + text.rstrip() + "|")         |   one piece|

text = "xoxo -one piece- xoox"
print("|" + text.strip(" xo") + "|")     |-one piece-|
print("|" + text.lstrip(" xo") + "|")     |-one piece- xoox|
print("|" + text.rstrip(" xo") + "|")     |xoxo -one piece-|

print(text.replace("xo", "*"))           ** -one piece- *ox

```


Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))  
print(text.startswith("Mi"))  
  
print(text.endswith("achi!"))  
print(text.endswith("Tachi!"))  
  
print(text.find("tachi"))  
print(text.rfind("tachi"))  
print(text.find("tacchi"))  
  
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True
```

```
print(text.startswith("Mi"))
```

```
print(text.endswith("achi!"))
```

```
print(text.endswith("Tachi!"))
```

```
print(text.find("tachi"))
```

```
print(text.rfind("tachi"))
```

```
print(text.find("tacchi"))
```

```
print(text.count("tac"))
```

```
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))          False
```

```
print(text.endswith("achi!"))  
print(text.endswith("Tachi!"))
```

```
print(text.find("tachi"))  
print(text.rfind("tachi"))  
print(text.find("tacchi"))
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))
```

```
print(text.find("tachi"))  
print(text.rfind("tachi"))  
print(text.find("tacchi"))
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))         False
```

```
print(text.find("tachi"))  
print(text.rfind("tachi"))  
print(text.find("tacchi"))
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))         False
```

```
print(text.find("tachi"))               10  
print(text.rfind("tachi"))  
print(text.find("tacchi"))
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))         False
```

```
print(text.find("tachi"))               10  
print(text.rfind("tachi"))              91  
print(text.find("tacchi"))
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))         False
```

```
print(text.find("tachi"))               10  
print(text.rfind("tachi"))              91  
print(text.find("tacchi"))              -1
```

```
print(text.count("tac"))  
print(text.count("tachi"))
```


Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))          False
```

```
print(text.endswith("achi!"))         True  
print(text.endswith("Tachi!"))        False
```

```
print(text.find("tachi"))              10  
print(text.rfind("tachi"))             91  
print(text.find("tacchi"))              -1
```

```
print(text.count("tac"))                8f  
print(text.count("tachi"))
```

Analyzing strings

```
text = """Ti che te tachi i tachi, tacame i me tachi.  
Mi no che no te taco i tachi, tachete ti i to tachi!"""
```

```
print(text.startswith("Ti"))           True  
print(text.startswith("Mi"))           False
```

```
print(text.endswith("achi!"))          True  
print(text.endswith("Tachi!"))         False
```

```
print(text.find("tachi"))               10  
print(text.rfind("tachi"))              91  
print(text.find("tacchi"))               -1
```

```
print(text.count("tac"))                 8  
print(text.count("tachi"))               5
```

Strings are immutable

Whenever you apply any of the operators or methods seen before, a new string is created. The original one is left unchanged.

```
>>> name = "alberto"
>>> othername = name.replace("al", "ro")
>>> together = name + othername
>>> print(name, othername, together)
alberto roberto albertoroberto
```

Strings are immutable

Differently from C/C++, but like Java, you cannot modify a character inside a string using the `[]` notation

```
>>> name[0] = "A"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item assignment
```

Example

Given an unformatted string of aminoacids, we want to remove the character >, remove spaces and convert everything to upper case

```
sequence = ">MANlFKLgaENIFLGrKW    "  
s1 = sequence.lstrip(">")  
s2 = s1.rstrip(" ")  
s3 = s2.upper()  
print(s3)
```

Alternatively

```
print(sequence.lstrip(">").rstrip(" ").upper())
```

Testing for equality

Result	Operator	Meaning
bool	==, !=	Check if two strings are equal or not
bool	is, not is	Check if two strings are identical or not
str	<, >	Check for lexicographic order

```
a1 = "casa"  
a2 = "casata"  
a3 = "casta"  
print(a1 == a2)  
print(a1 < a2)  
print(a1 < a3)
```

Testing for equality

Result	Operator	Meaning
bool	==, !=	Check if two strings are equal or not
bool	is, not is	Check if two strings are identical or not
str	<, >	Check for lexicographic order

```
a1 = "casa"
```

```
a2 = "casata"
```

```
a3 = "casta"
```

```
print(a1 == a2)
```

```
False
```

```
print(a1 < a2)
```

```
True
```

```
print(a1 < a3)
```

```
True
```

Operator ==: equality

Python

```
a1 = "banana"
a2 = "banana"
b1 = "ba"+"na"
b2 = "ba"+"na"
c1 = b1+"na"
c2 = b2+"na"
print(a1==a2)
print(b1==b2)
print(c1==c2)
```

True
True
True

Java

```
String a1 = "banana";
String a2 = "banana";
String b1 = "ba"+"na";
String b2 = "ba"+"na";
String c1 = b1+"na";
String c2 = b2+"na";
System.out.println(a1 == a2);
System.out.println(b1 == b2);
System.out.println(c1 == c2);
```

true
true
false

Operator `is`: identity

Python

```
a1 = "banana"
a2 = "banana"
b1 = "ba"+"na"
b2 = "ba"+"na"
c1 = b1+"na"
c2 = b2+"na"
print(a1 is a2)
print(b1 is b2)
print(c1 is c2)
```

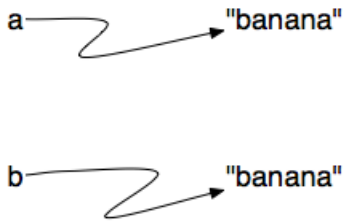
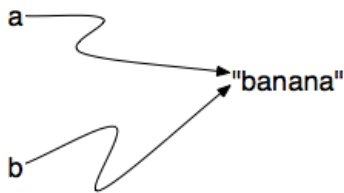
```
True
True
False
```

Java

```
String a1 = "banana";
String a2 = "banana";
String b1 = "ba"+"na";
String b2 = "ba"+"na";
String c1 = b1+"na";
String c2 = b2+"na";
System.out.println(a1.equals(a2));
System.out.println(b1.equals(b2));
System.out.println(c1.equals(c2));
```

```
true
true
true
```

How strings are stored



In one case, **a** and **b** refer to two different string objects that have the same value. In the second case, they refer to the same object. Remember that an object is something a variable can refer to.

Exercises

- Check whether a string contains exactly five (arbitrary) characters
- Check whether a string contains at least one space
- Check whether the string "12345" begins with 1
- Check whether a string contains `x` at least three times at the beginning and/or at the end. For instance, the following strings satisfy the desideratum: `"x...xx"`, `"xx...x"`, `"xxxx..."`.

Table of contents

- 1 Strings
- 2 Lists**
- 3 Tuples
- 4 Dictionaries

Lists

Lists

Lists are ordered sequences of arbitrary elements (objects).

- Lists are **mutable**: it is possible to change an element inside a list.
- Lists are a **sequential collections** of elements. This means that the individual elements that make up the list are assumed to be in a particular order from left to right.
- A list that contains no element, often referred to as the **empty list**, is still considered to be a list.

How to define lists

Lists are defined using square brackets, as follows:

```
# A list of integers (notice that the 1 appears twice)
```

```
integers = [1, 2, 3, 1]
```

```
# A list of strings
```

```
uniprot_proteins = ["Y08501", "Q95747"]
```

```
# A list of heterogeneous objects
```

```
things = ["Y08501", 0.13, "Q95747", 0.96]
```

```
# An empty list
```

```
empty = []
```

How to define lists

```
# A list of lists
```

```
two_level_list = [  
    ["Y08501", 120, 520],  
    ["Q95747", 550, 920],  
]
```

```
# A list containing two empty lists
```

```
a_weird_list = [ [], [] ]
```

List operators

All these operators work exactly as in strings

Result	Operator	Meaning
bool	<code>==, !=</code>	Check if two lists are equal or different
int	<code>len(list)</code>	Return the length of the list
list	<code>list + list</code>	Concatenate two lists (returns a new list)
list	<code>list * int</code>	Replicate the list (returns a new list)
list	<code>list[int:int]</code>	Extract a sub-list

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

True

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

True / False

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

True / False / False

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

True / False / False / False

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```
food = ["apple", "orange", "banana", "cherry",  
        ["blueberry", "strawberry", "raspberry"]]  
print("apple" in food)  
print("pear" in food)  
print([] in food)  
print(["apple", "orange"] in food)  
print("blueberry" in food)  
print(["blueberry", "strawberry", "raspberry"] in food)
```

True / False / False / False / False

New and slightly different operators

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

```

food = ["apple", "orange", "banana", "cherry",
        ["blueberry", "strawberry", "raspberry"]]
print("apple" in food)
print("pear" in food)
print([] in food)
print(["apple", "orange"] in food)
print("blueberry" in food)
print(["blueberry", "strawberry", "raspberry"] in food)

```

True / False / False / False / False / True

New and slightly different operators

Result	Operator	Meaning
obj	list[int]	Read/write an element at a specified index

```
food = ["apple", "orange", "banana", "cherry"]
food[1]="pear"
print(food[1])
print(food[2]=="banana")
food[4] = "pineapple"
```


New and slightly different operators

Result	Operator	Meaning
obj	list[int]	Read/write an element at a specified index

```

food = ["apple", "orange", "banana", "cherry"]
food[1]="pear"
print(food[1])
print(food[2]=="banana")
food[4] = "pineapple"

```

```

pear
True

```

```

Traceback (most recent call last):

```

```

  File "<stdin>", line 1, in <module>

```

```

IndexError: list assignment index out of range

```

Notes on lists

Lists are ordered

```
[1,2,3] != [3,2,1]
```

Lists are not sets

```
[3, 3, "a", "a"] != [3, "a"]
```

Matrix

```

matrix = [
    [1, 2, 3],           # <-- 1st row
    [4, 5, 6],           # <-- 2nd row
    [7, 8, 9],           # <-- 3rd riga
]
#   ^   ^   ^
#   |   |   |
#   |   |   +-- 3rd column
#   |   +----- 2nd column
#   +----- 1st column

print(matrix[0])

```

Matrix

```

matrix = [
    [1, 2, 3],           # <-- 1st row
    [4, 5, 6],           # <-- 2nd row
    [7, 8, 9],           # <-- 3rd riga
]
#   ^   ^   ^
#   |   |   |
#   |   |   +-- 3rd column
#   |   +----- 2nd column
#   +----- 1st column

print(matrix[0])        [1, 2, 3]
print(matrix[1][1])

```

Matrix

```

matrix = [
    [1, 2, 3],           # <-- 1st row
    [4, 5, 6],           # <-- 2nd row
    [7, 8, 9],           # <-- 3rd riga
]
#   ^   ^   ^
#   |   |   |
#   |   |   +-- 3rd column
#   |   +----- 2nd column
#   +----- 1st column

print(matrix[0])        [1, 2, 3]
print(matrix[1][1])     5
print(matrix[-1][-1])

```

Matrix

```

matrix = [
    [1, 2, 3],           # <-- 1st row
    [4, 5, 6],           # <-- 2nd row
    [7, 8, 9],           # <-- 3rd riga
]
#   ^   ^   ^
#   |   |   |
#   |   |   +-- 3rd column
#   |   +----- 2nd column
#   +----- 1st column

print(matrix[0])         [1, 2, 3]
print(matrix[1][1])     5
print(matrix[-1][-1])   9

```

List methods

Return	Method	Meaning
None	<code>list.append(obj)</code>	Add a new element at the end of the list
None	<code>list.extend(list)</code>	Add several new elements at the end of the list
None	<code>list.insert(int,obj)</code>	Add a new element at some given position
None	<code>list.remove(obj)</code>	Remove the first occurrence of an element
None	<code>list.reverse()</code>	Invert the order of the elements
None	<code>list.sort()</code>	Sort the elements
int	<code>list.count(obj)</code>	Count the occurrences of an element

List methods

```
L = [1,2,3]
print(L)
L.append(4)
print(L)
L.extend([7,6,5])
print(L)
L.insert(3, 3.5)
print(L)
L.remove(3.5)
print(L)
L.sort()
print(L)
L.reverse()
print(L)
```


List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)
L.extend([7,6,5])
print(L)
L.insert(3, 3.5)
print(L)
L.remove(3.5)
print(L)
L.sort()
print(L)
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)
L.insert(3, 3.5)
print(L)
L.remove(3.5)
print(L)
L.sort()
print(L)
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.insert(3, 3.5)
print(L)
L.remove(3.5)
print(L)
L.sort()
print(L)
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.insert(3, 3.5)
print(L)           [1, 2, 3, 3.5, 4, 7, 6, 5]
L.remove(3.5)
print(L)
L.sort()
print(L)
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.insert(3, 3.5)
print(L)           [1, 2, 3, 3.5, 4, 7, 6, 5]
L.remove(3.5)
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.sort()
print(L)
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.insert(3, 3.5)
print(L)           [1, 2, 3, 3.5, 4, 7, 6, 5]
L.remove(3.5)
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.sort()
print(L)           [1, 2, 3, 4, 5, 6, 7]
L.reverse()
print(L)
```

List methods

```
L = [1,2,3]
print(L)           [1, 2, 3]
L.append(4)
print(L)           [1, 2, 3, 4]
L.extend([7,6,5])
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.insert(3, 3.5)
print(L)           [1, 2, 3, 3.5, 4, 7, 6, 5]
L.remove(3.5)
print(L)           [1, 2, 3, 4, 7, 6, 5]
L.sort()
print(L)           [1, 2, 3, 4, 5, 6, 7]
L.reverse()
print(L)           [7, 6, 5, 4, 3, 2, 1]
```

List methods

All list methods (except `count()`):

- Modify the input list
- Do not have a return value (they return `None`)

```
L = [0, 1, 2, 3, 4, 5]
```

```
print(L)
```

```
result = L.append(6)
```

```
print(L)
```

```
print(result)
```

```
result.append(7)
```


List methods

All list methods (except `count()`):

- Modify the input list
- Do not have a return value (they return `None`)

```
L = [0, 1, 2, 3, 4, 5]
```

```
print(L)                [0, 1, 2, 3, 4, 5]
```

```
result = L.append(6)
```

```
print(L)
```

```
print(result)
```

```
result.append(7)
```

List methods

All list methods (except `count()`):

- Modify the input list
- Do not have a return value (they return `None`)

```
L = [0, 1, 2, 3, 4, 5]
```

```
print(L)                [0, 1, 2, 3, 4, 5]
```

```
result = L.append(6)
```

```
print(L)                [0, 1, 2, 3, 4, 5, 6]
```

```
print(result)
```

```
result.append(7)
```

List methods

All list methods (except `count()`):

- Modify the input list
- Do not have a return value (they return `None`)

```
L = [0, 1, 2, 3, 4, 5]
```

```
print(L)           [0, 1, 2, 3, 4, 5]
```

```
result = L.append(6)
```

```
print(L)           [0, 1, 2, 3, 4, 5, 6]
```

```
print(result)      None
```

```
result.append(7)
```

List methods

All list methods (except `count()`):

- Modify the input list
- Do not have a return value (they return `None`)

```
L = [0, 1, 2, 3, 4, 5]
```

```
print(L)           [0, 1, 2, 3, 4, 5]
```

```
result = L.append(6)
```

```
print(L)           [0, 1, 2, 3, 4, 5, 6]
```

```
print(result)      None
```

```
result.append(7)   Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object
has no attribute 'append'
```

Compare them with similar methods for strings

Append, extend vs concatenation

Unless you really want to create a new list, avoid concatenation whenever possible. The following code produces the same result, but concatenation is way more inefficient.

```
a = [0,1,2,3,4,5]
a.append(6)
a.extend([7,8,9])
```

```
a = [0,1,2,3,4,5]
a = a + [6]
a = a + [7,8,9]
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
L1 = [1,2,3]
L2 = [4,5]
LL = [L1,L2]
print(LL)
L1[2] = 10
print(LL)
L2.append(6)
print(LL)
LL[1][1] = 0
print(L2)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
L1 = [1,2,3]
L2 = [4,5]
LL = [L1,L2]
print(LL)                [[1, 2, 3], [4, 5]]
L1[2] = 10
print(LL)
L2.append(6)
print(LL)
LL[1][1] = 0
print(L2)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
L1 = [1,2,3]
L2 = [4,5]
LL = [L1,L2]
print(LL)                [[1, 2, 3], [4, 5]]
L1[2] = 10
print(LL)                [[1, 2, 10], [4, 5]]
L2.append(6)
print(LL)
LL[1][1] = 0
print(L2)
```


Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
L1 = [1,2,3]
L2 = [4,5]
LL = [L1,L2]
print(LL)           [[1, 2, 3], [4, 5]]
L1[2] = 10
print(LL)           [[1, 2, 10], [4, 5]]
L2.append(6)
print(LL)           [[1, 2, 10], [4, 5, 6]]
LL[1][1] = 0
print(L2)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
L1 = [1,2,3]
L2 = [4,5]
LL = [L1,L2]
print(LL)           [[1, 2, 3], [4, 5]]
L1[2] = 10
print(LL)           [[1, 2, 10], [4, 5]]
L2.append(6)
print(LL)           [[1, 2, 10], [4, 5, 6]]
LL[1][1] = 0
print(L2)           [4, 0, 6]
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4]
copy = original
copy.append(5)
print(original)
print(copy)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4]
copy = original
copy.append(5)
print(original)           [1, 2, 3, 4, 5]
print(copy)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4]
copy = original
copy.append(5)
print(original)      [1, 2, 3, 4, 5]
print(copy)          [1, 2, 3, 4, 5]
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4,5]
copy = original[:]
copy.append(6)
print(original)
print(copy)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4,5]
copy = original[:]
copy.append(6)
print(original)           [1, 2, 3, 4, 5]
print(copy)
```

Consequences of mutability

Recall that lists are mutable, and that (like all variables) they contain references to objects, not the objects themselves.

```
original = [1,2,3,4,5]
copy = original[:]
copy.append(6)
print(original)      [1, 2, 3, 4, 5]
print(copy)          [1, 2, 3, 4, 5, 6]
```


Equality and identity

```
a = [81, 82, 83]
b = [81, 82, 83]
print(a is b)
print(a == b)
```

Equality and identity

```
a = [81, 82, 83]
```

```
b = [81, 82, 83]
```

```
print(a is b)
```

False

```
print(a == b)
```

Equality and identity

```
a = [81, 82, 83]
b = [81, 82, 83]
print(a is b)           False
print(a == b)          True
```

Even though they are initialized with the same value, the two objects references by variables **a** and **b** are not identical.

Why this difference with strings?

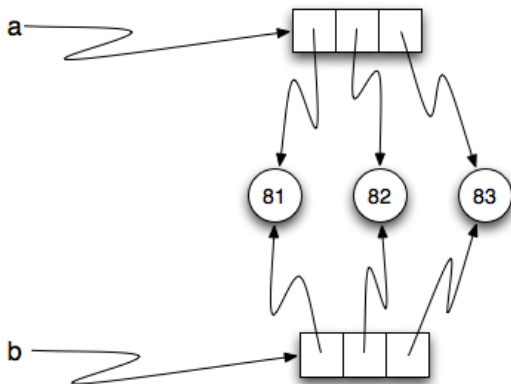
Equality and identity

```
a = [81, 82, 83]
b = [81, 82, 83]
print(a is b)           False
print(a == b)          True
```

Even though they are initialized with the same value, the two objects references by variables **a** and **b** are not identical.

Why this difference with strings? **Lists are mutable**

Equality and identity



Equality and identity

```
a = [81, 82, 83]
b = a
c = [81, 82, 83]
b[1] = 85
print(a)
print(b)
print(c)
```

Equality and identity

```
a = [81, 82, 83]
b = a
c = [81, 82, 83]
b[1] = 85
print(a)
print(b)
print(c)
```

Equality and identity

```
a = [81, 82, 83]
b = a
c = [81, 82, 83]
b[1] = 85
print(a)           [81, 85, 83]
print(b)
print(c)
```

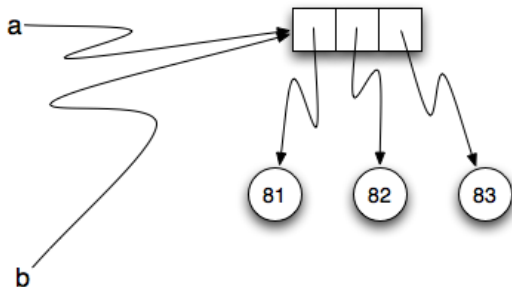

Equality and identity

```
a = [81, 82, 83]
b = a
c = [81, 82, 83]
b[1] = 85
print(a)           [81, 85, 83]
print(b)           [81, 85, 83]
print(c)
```

Equality and identity

```
a = [81, 82, 83]
b = a
c = [81, 82, 83]
b[1] = 85
print(a)           [81, 85, 83]
print(b)           [81, 85, 83]
print(c)           [81, 82, 83]
```

Equality and identity



Exotic slicing

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = [7, 8]
```

```
print(L)
```

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = []
```

```
print(L)
```

```
L = [1, 4, 6]
```

```
L[1:1] = [2, 3]
```

```
print(L)
```

```
L[4:4] = [5]
```

```
print(L)
```

Exotic slicing

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = [7, 8]
```

```
print(L)
```

```
[1, 7, 8, 4, 5, 6]
```

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = []
```

```
print(L)
```

```
L = [1, 4, 6]
```

```
L[1:1] = [2, 3]
```

```
print(L)
```

```
L[4:4] = [5]
```

```
print(L)
```

Exotic slicing

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = [7, 8]
```

```
print(L)
```

```
[1, 7, 8, 4, 5, 6]
```

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = []
```

```
print(L)
```

```
[1, 4, 5, 6]
```

```
L = [1, 4, 6]
```

```
L[1:1] = [2, 3]
```

```
print(L)
```

```
L[4:4] = [5]
```

```
print(L)
```

Exotic slicing

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = [7, 8]
```

```
print(L)
```

```
[1, 7, 8, 4, 5, 6]
```

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = []
```

```
print(L)
```

```
[1, 4, 5, 6]
```

```
L = [1, 4, 6]
```

```
L[1:1] = [2, 3]
```

```
print(L)
```

```
[1, 2, 3, 4, 6]
```

```
L[4:4] = [5]
```

```
print(L)
```

Exotic slicing

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = [7, 8]
```

```
print(L)
```

```
[1, 7, 8, 4, 5, 6]
```

```
L = [1, 2, 3, 4, 5, 6]
```

```
L[1:3] = []
```

```
print(L)
```

```
[1, 4, 5, 6]
```

```
L = [1, 4, 6]
```

```
L[1:1] = [2, 3]
```

```
print(L)
```

```
[1, 2, 3, 4, 6]
```

```
L[4:4] = [5]
```

```
print(L)
```

```
[1, 2, 3, 4, 5, 6]
```


String-list methods

Result	Operator	Meaning
list-of-str	<code>str.split(str)</code>	Split a string into a list of strings (words)

`s.split(sep)` returns a list of the words of the string `s`. If the optional argument `sep` is absent or `None`, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). Otherwise, `sep` specifies a string to be used as the word separator.

String-list methods

```
terzina = """Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
che la diritta via era smarrita."""
versi = terzina.split("\n")
print(versi)
print(versi[0].split())
print(versi[1].split(" "))
print(versi[2].split("via"))
```

String-list methods

```
terzina = """Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
che la diritta via era smarrita."""
versi = terzina.split("\n")
print(versi)
print(versi[0].split())
print(versi[1].split(" "))
print(versi[2].split("via"))
```

```
['Nel mezzo del cammin di nostra vita',
'mi ritrovai per una selva oscura,',
'che la diritta via era smarrita.']
['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']
['mi', 'ritrovai', 'per', 'una', 'selva', 'oscura,']
['che la diritta ', ' era smarrita.']
```

Exercises

What is the difference between these two pieces of code? How long is `list` in the two cases?

```
L = []  
L.append([1,2,3])  
L.append([4,5,6])  
print(L)  
print(len(L))
```

```
L = []  
L.extend([1,2,3])  
L.extend([4,5,6])  
print(L)  
print(len(L))
```

Exercises

What is the difference between these two pieces of code? How long is `list` in the two cases?

```
L = []
L.append([1,2,3])
L.append([4,5,6])
print(L)
print(len(L))
```

```
[ [1, 2, 3], [4, 5, 6] ]
2
```

```
L = []
L.extend([1,2,3])
L.extend([4,5,6])
print(L)
print(len(L))
```

```
[1,2,3,4,5,6]
6
```

Table of contents

- 1 Strings
- 2 Lists
- 3 Tuples**
- 4 Dictionaries

Tuples

Tuples

Tuples are the immutable version of lists.

- Tuples are **immutable**: it is not possible to change an element inside a tuple.
- Tuples are a **sequential collections** of elements. This means that the individual elements that make up the tuple are assumed to be in a particular order from left to right.
- Tuples with zero or one elements are possible, but not really interesting.

How to define tuples

Tuples are defined using paranthesis, as follows:

```
# A tuple of integers (notice that the 1 appears twice)
```

```
integers = (1, 2, 3, 1)
```

```
# A tuple of strings
```

```
uniprot_proteins = ("Y08501", "Q95747")
```

```
# A tuple of heterogeneous objects
```

```
things = ("Y08501", 0.13, "Q95747", 0.96)
```

```
# This is not a tuple, is a variable initialized to
```

```
# an expression evaluated to 1
```

```
single = (1)
```


How to define tuples

```
# A tuple containing a single element
```

```
single = (1,)
```

```
# A tuple of tuples
```

```
two_level_list = (  
    ("Y08501", 120, 520),  
    ("Q95747", 550, 920),  
)
```

Why tuples?

- Tuples are needed whenever an immutable version of lists is needed. E.g., tuples can be used as **keys** in *dictionaries*, yet another data structure that associates immutable keys to objects.
- Tuples are used to associate objects that are treated as a single entity in the program. E.g., functions may return tuples in order to return multiple objects at the same time.
- Whenever a sequence of objects cannot change over time, immutable tuples are more efficient than mutable lists.

Tuple operators

All these operators work exactly as in lists

Result	Operator	Meaning
bool	<code>==, !=</code>	Check if two tuples are equal or different
int	<code>len(tuple)</code>	Return the length of the tuple
tuple	<code>tuple + tuple</code>	Concatenate two tuples (returns a new tuple)
tuple	<code>tuple * int</code>	Replicate the tuple (returns a tuple)
tuple	<code>tuple[int]</code>	Read an element of the tuple
tuple	<code>tuple[int:int]</code>	Extract a sub-tuple

Tuple methods

Return	Method	Meaning
int	<code>tuple.count(obj)</code>	Count the occurrences of an element
int	<code>tuple.index(obj)</code>	Return the index of the first occurrence of an object

Some comments

- Equality and identity for tuples work exactly as equality and identity for lists, not as in strings
- As tuples are immutable, there are no consequences of mutability (unlike lists)

List/tuple/string conversions

Return	Method	Meaning
list	list(obj)	Transform an object into a list
tuple	tuple(obj)	Transform an object into a tuple

```
T = (1,2,3)
S = "123"
LT = list(T)
LS = list(S)
print(LT)
print(LS)
print(LT == LS)
```

List/tuple/string conversions

Return	Method	Meaning
list	list(obj)	Transform an object into a list
tuple	tuple(obj)	Transform an object into a tuple

```
T = (1,2,3)
```

```
S = "123"
```

```
LT = list(T)
```

```
LS = list(S)
```

```
print(LT)
```

```
[1, 2, 3]
```

```
print(LS)
```

```
['1', '2', '3']
```

```
print(LT == LS)
```

```
False
```

Warning

Why you shouldn't call a list `list`, a string `string` and a tuple `tuple`

```
list = [1,2,3]
another_list = list("Goal")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
```

Your variable `list` substitutes the `list()` function; more on this in a future lecture.

Table of contents

- 1 Strings
- 2 Lists
- 3 Tuples
- 4 Dictionaries**

Dictionary

Dictionary

A dictionary represents a map between objects: it maps from a key to the corresponding value.

- Dictionaries are **mutable**: it is possible to add/remove/change the associations between keys and values
- Dictionaries contains sequences of keys, but these keys are not necessarily ordered.
- A dictionary that contains no element is still considered to be a dictionary.

How to define dictionaries

Dictionaries are defined using curly brackets, listing associations
key:values: key1: value1, key2: value2, ...

```
genetic_code = {  
    "UUU": "F",      # phenilalanyne  
    "UCU": "S",      # serine  
    "UAU": "Y",      # tyrosine  
    "UGU": "C",      # cysteine  
    "UUC": "F",      # phenilalanyne  
    "UCC": "S",      # serine  
    "UAC": "Y",      # tyrosine  
    # etc.  
}
```

- **Keys** are unique: a key can be associated to a single value
- **Values** are not unique: different keys can map to the same value

How to define dictionaries

```
volume_of = {  
    "A": 67.0, "C": 86.0, "D": 91.0,  
    "E": 109.0, "F": 135.0, "G": 48.0,  
    "H": 118.0, "I": 124.0, "K": 135.0,  
    "L": 124.0, "M": 124.0, "N": 96.0,  
    "P": 90.0, "Q": 114.0, "R": 148.0,  
    "S": 73.0, "T": 93.0, "V": 105.0,  
    "W": 163.0, "Y": 141.0,  
}
```

- There are no restrictions on the type of the values
- In this case, values are floats

Reading a dictionary

```
>>> print(genetic_code["UCU"])
S
>>> print(genetic_code["UCC"])
S
>>> print(volume_of["C"])
86.0
>>> print(type(volume_of["C"]))
float
```

Keys must be immutable

The association works only in one direction: you can obtain values from keys, not viceversa

```
properties_of = {
    "A": [ 89.09,  67.0],
    "C": [121.15,  86.0],
    "D": [133.10,  91.0],
    # ...
}
print(properties_of["A"])
print(properties_of[89.09,  67.0])
```

Keys must be immutable

The association works only in one direction: you can obtain values from keys, not viceversa

```
properties_of = {  
    "A": [ 89.09,  67.0],  
    "C": [121.15,  86.0],  
    "D": [133.10,  91.0],  
    # ...  
}  
  
print(properties_of["A"])  
print(properties_of[89.09,  67.0])
```

```
[89.09, 67.0]
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
KeyError: (89.09, 67.0)
```

Keys must be immutable

Lists and dictionaries cannot be used as keys, because they are mutable.

```
reverse_properties_of = {  
    [89.09, 67.0]: "A",  
    [121.15, 86.0]: "C",  
    [133.10, 91.0]: "D",  
    # ...  
}
```

Traceback (most recent call last):

```
File "<stdin>", line 4, in <module>  
TypeError: unhashable type: 'list'
```


Keys must be immutable

Use tuples instead!

```
reverse_properties_of = {  
    (89.09, 67.0): "A",  
    (121.15, 86.0): "C",  
    (133.10, 91.0): "D",  
    # ...  
}
```

Dictionary operators

Result	Operator	Meaning
bool	obj in dict	Return True if a key is present in the dictionary
int	len(dict)	Return the number of elements in the dictionary
obj	dict[obj]	Read the value associate with a key
obj	dict[obj] = obj	Add or modify the value associated with a key

```
code = {}           # Empty dictionary
code["UUU"] = "F"  # Phenylalanine
code["UCU"] = "M"  # Methionine
code["UCU"] = "S"  # Serine (methionine was a mistake!)
code["UAU"] = "Y"  # Tyrosine
print(len(code))
print("Y" in code)
```

Dictionary operators

Result	Operator	Meaning
bool	obj in dict	Return True if a key is present in the dictionary
int	len(dict)	Return the number of elements in the dictionary
obj	dict[obj]	Read the value associate with a key
obj	dict[obj] = obj	Add or modify the value associated with a key

```
code = {}           # Empty dictionary
code["UUU"] = "F"  # Phenylalanine
code["UCU"] = "M"  # Methionine
code["UCU"] = "S"  # Serine (methionine was a mistake!)
code["UAU"] = "Y"  # Tyrosine
print(len(code))   # 3
print("Y" in code) # False
```

Dictionary methods

Return	Method	Meaning
list	<code>dict.keys()</code>	Returns the list of the keys that are present in the dictionary
list	<code>dict.values()</code>	Returns the list of the values that are present in the dictionary
list of tuples	<code>list.items()</code>	Returns the list of pairs (key, value) that are present in the dictionary

Dictionary methods

```
code = {  
    "UUU": "F",      # phenylalanine  
    "UCU": "S",      # serine  
    "UAU": "Y",      # tyrosine  
    "UGU": "C",      # cysteine  
    "UUC": "F",      # phenylalanine  
    "UCC": "S",      # serine  
    "UAC": "Y",      # tyrosine  
}
```

Dictionary methods

```
>>> print(code)
{'UCC': 'S', 'UCU': 'S', 'UUC': 'F', 'UUU': 'F', 'UGU': 'C',
 'UAC': 'Y', 'UAU': 'Y'}
```

```
>>> print(code.keys())
dict_keys(['UCU', 'UAC', 'UUU', 'UUC', 'UGU', 'UAU', 'UCC'])
```

```
>>> print(code.values())
dict_values(['S', 'Y', 'F', 'F', 'C', 'Y', 'S'])
```

```
>>> print(code.items())
dict_items([('UCU', 'S'), ('UAC', 'Y'), ('UUU', 'F'),
 ('UUC', 'F'), ('UGU', 'C'), ('UAU', 'Y'), ('UCC', 'S')])
```

Dictionary methods

```
>>> print(code)
{'UCC': 'S', 'UCU': 'S', 'UUC': 'F', 'UUU': 'F', 'UGU': 'C',
 'UAC': 'Y', 'UAU': 'Y'}
```

```
>>> print(list(code.keys()))
['UUU', 'UGU', 'UUC', 'UAU', 'UCU', 'UAC', 'UCC']
```

```
>>> print(list(code.values()))
['F', 'C', 'F', 'Y', 'S', 'Y', 'S']
```

```
>>> print(list(code.items()))
[('UUU', 'F'), ('UGU', 'C'), ('UUC', 'F'), ('UAU', 'Y'),
 ('UCU', 'S'), ('UAC', 'Y'), ('UCC', 'S')]
```

Notes

- Associations are stored (and printed) in a random order, which is neither the order in which elements are inserted neither the alphabetical order
- Differences between 2.x and 3.x:
 - In 2.x, methods `keys()`, `values()` and `items()` returns lists
 - In 3.x, methods `keys()`, `values()` and `items()` return special iterable objects used in `for` loops

Example

```
seq = "GTCCCTGTTCGGGCGCCA"  
num_A = seq.count("A")  
num_T = seq.count("T")  
num_C = seq.count("C")  
num_G = seq.count("G")  
histogram = {  
    "A": num_A / len(seq),  
    "T": num_T / len(seq),  
    "C": num_C / len(seq),  
    "G": num_G / len(seq),  
}  
print(histogram)
```