# Fundamentals of Information Systems

## Python Programming (for Data Science)

### Master's Degree in Data Science

**Gabriele Tolomei**

gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
October 18, 2018

# Lecture 3: Python's Built-in Data Types (1)

# Data Type Hierarchy

- Python's built-in data types can be grouped into several classes.
- We use the same hierarchy scheme used in the official Python documentation, which defines the following classes:
    - **numeric**, **sequences**, **sets** and **mappings** (and a few more not discussed further here).
- A special mention goes to two particular data types: `bool` and `NoneType`.

# Booleans

# Type `bool` (*immutable*)

- It encapsulates the two boolean values which are written as `True` and `False`.
- Comparisons and other conditional expressions evaluate to either `True` or `False`.
- Boolean values are combined with the `and` and `or` keywords.

In [1]:

```python
type(True)
```

Out[1]: bool

# Boolean Operations: `or`, `and`, `not`

- Ordered by ascending priority:

| Operation | Result |
|---|---|
| `x or y` | if *x* is false, then *y*, else *x* |
| `x and y` | if *x* is false, then *x*, else *y* |
| `not x` | if *x* is false, then `True`, else `False` |

In [2]:

```
False or True
```

Out[2]: True

In [3]:

```
True and True
```

Out[3]:

```
True
```

# Comparisons

- There are **eight** comparison operations in Python.
- They all have the same priority (which is higher than that of the Boolean operations).
- Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated **only once** (but in both cases `z` is not evaluated at all when `x < y` is found to be `False`).

# Table of Comparisons

| Operation | Meaning |
|---|---|
| `<` | strictly less than |
| `<=` | less than or equal |
| `>` | strictly greater than |
| `>=` | greater than or equal |
| `==` | equal |
| `!=` | not equal |
| `is` | object identity |
| `is not` | negated object identity |

# A Quick Note on the `is` Operator

- It is used to compare the **identity** of two objects.
- The **identity** of an object can be found with the `id()` built-in function.
- `id()` takes as input a Python object and returns an integer representing the identity of *that* object.
- In the standard CPython implementation, this integer corresponds to the object's location in memory (in other implementations/platforms this might be different).

# `is` *vs.* `==`

- `is` is used to test for **identity** of two objects by means of the `id()` function.
- `==` is used to test for the **value** of two objects.
- In other words, if you have 2 objects `x` and `y` the statement below

  ```
  x is y
  ```

  corresponds to the following:

  ```
  id(x) == id(y)
  ```

In [4]:

```python
# Using the 'is' operator in combination with immutable objects (e.g., integers)
x = 42
y = x
print("id(x) = {}".format(id(x)))
print("id(42) = {}".format(id(42)))
print("id(y) = {}".format(id(y)))
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y
# Modifying x (immutable) means creating a new integer object and assign it to x
x += 1
print("id(x) = {}".format(id(x)))
print("id(43) = {}".format(id(43)))
print("id(42) = {}".format(id(42)))
print("id(y) = {}".format(id(y)))
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y
```

```
id(x) = 4460251760
id(42) = 4460251760
id(y) = 4460251760
Q: The identity of x is the same of that of y? A: True
id(x) = 4460251792
id(43) = 4460251792
id(42) = 4460251760
id(y) = 4460251760
Q: The identity of x is the same of that of y? A: False
```

In [5]:

```python
# Using the 'is' operator in combination with mutable objects (e.g., lists)
x = [1, 2, 3]
y = x
print("id(x) = {}".format(id(x)))
print("id(y) = {}".format(id(y)))
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y
# Let's modify x
x.append(4)
print("id(x) = {}".format(id(x)))
print("id(y) = {}".format(id(y)))
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y
```

```
id(x) = 4502699464
id(y) = 4502699464
Q: The identity of x is the same of that of y? A: True
id(x) = 4502699464
id(y) = 4502699464
Q: The identity of x is the same of that of y? A: True
```

In [6]:

```python
# Unexpected behaviors which might cause you some problems...
x = 42
y = 42
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y,
print("Q: The value of x is the same of that of y? A: {}".format(x == y))
x = 257
y = 257
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y,
print("Q: The value of x is the same of that of y? A: {}".format(x == y))
# This odd behavior depends on the fact that CPython implements
# integers in the range (-5, 256) at fixed memory locations. As such, any named variable
# referencing one of those integers will always have the same memory address.
# On the other hand, integers outside that range might be possibly allocated at different
# memory addresses and therefore they have different identities even though the same value!
# Long story short, if you want to test for equality DO USE '=='
```

```
Q: The identity of x is the same of that of y? A: True
Q: The value of x is the same of that of y? A: True
Q: The identity of x is the same of that of y? A: False
Q: The value of x is the same of that of y? A: True
```

In [7]:

```python
# When you work with mutable objects you will always face the following behavior
x = [1, 2, 3]
y = [1, 2, 3] # Note that here we are assigning a 'new' object to y
print("Q: The identity of x is the same of that of y? A: {}".format(x is y))# id(x) == id(y
print("Q: The value of x is the same of that of y? A: {}".format(x == y))
```

```
Q: The identity of x is the same of that of y? A: False
Q: The value of x is the same of that of y? A: True
```

# Non-zero Interpretation

- Almost all built-in Python types (and any class defining the `__nonzero__` method) have a `True` or `False` interpretation in an `if` statement

In [8]:

```python
x = [1, 2, 3] # define a list with 3 elements
if x:
    print('The list contains something!')

y = [] # define an empty list
if not y:
    print('The list is empty!')
```

```
The list contains something!
The list is empty!
```

# True- or Falseness

- Most objects in Python have a notion of true- or falseness.
- For example, empty sequences like lists, dicts, tuples, etc. (more on those types later on) are treated as `False` if used in control flow (see the empty list `y` above).
- You can see exactly what boolean value an object coerces to by invoking `bool` on it.

In [9]:

```
bool([]), bool([1, 2, 3])
```

Out[9]:

```
(False, True)
```

In [10]:

```python
bool('Hello World!'), bool('')
```

Out[10]:

```
(True, False)
```

In [11]:
```python
bool(0), bool(1)
```
Out[11]:
```
(False, True)
```

None

# Type `NoneType` (*immutable*) and `None` instance

- `None` is the Python **null** value type.
- Actually, it is the unique available *instance* of `NoneType` object.
- If a function does not explicitly return a value, it implicitly returns `None`.
- `None` is also a common default value for *optional* function arguments.

In [12]:

```
a = None
a is None
```

Out[12]: True

In [13]:

```
b = 42
b is None
```

Out[13]: False

In [14]:

```python
# z is an optional input argument of the following function
def add_and_possibly_multiply(x, y, z=None):

    result = x + y # sum the first two positional input arguments

    if z is not None: # multiply the current result by z iff z is not None
        result *= z

    return result # finally, return the result
```

# Numerics

# Numeric Types: `int`, `float`, `complex` (*immutables*)

- The primary Python types for numbers are:
  - `int`: represents arbitrarily large integers (in Python 2.x this is equivalent to C `long`);
  - `float`: floating-point numbers (equivalent to 64-bit C `double`);
  - `complex`: complex numbers.

In [15]:

```python
# An integer number
ix = 123456789
# A very large integer obtained from the one before by rising it to the 8-th power
ix ** 8
```

Out[15]: 53965948844821664748141453212125737955899777414752273389058576481

# A Quick Note on Extremely Large Integers

- On Python 2.x, `sys.maxint` gives you the (maximum) integer value which your computer can work **natively** with.
- Up to `sys.maxint` your machine is able to perform arithmetic operation (e.g., addition, multiplication) in a **single** CPU instruction.
- This value corresponds to the number that can be represented using 64 bits (if your platform word's size is 64 bits, otherwise 32 bits, etc.).

# A Quick Note on Extremely Large Integers: Beyond `sys.maxint`

- Just because that is what can be done in a single CPU instruction does not mean you cannot go beyond that limit!
- Python introduces **extended-precision integers** to overcome such a limitation.
- Those are "sofware structures" that can handle integers of any size transparently to the user by chaining them together, only limited by the memory available.
- Python 2.x keeps native integers "separate" from extended-precision ones, whilst Python 3.x treats every integer as extended-precision.

In [16]:

```python
# A float number
fx = 3.645
# A float number defined using scientific notation
fx_exp = 8.21e-4
```

# A Quick Note of Floating-Point Arithmetic

- As opposed to integers, floating-point numbers have a finite-precision representation in computer hardware as base 2 (binary) fractions.
- For example, consider the decimal fraction 0.125 and the binary fraction 0.001
- Both represent the same number: $1 * 10^{-1} + 2 * 10^{-2} + 5 * 10^{-3} = 0 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$
- Unfortunately, most decimal fractions cannot be represented *exactly* as binary fractions.
- As such, decimal floating-point numbers are thus approximated by the binary floating-point numbers actually stored in the machine.

# A Quick Note of Floating-Point Arithmetic: Issues

- No matter how many decimal digits you use, you will not get the exact representation of the fraction 1/3 = 0.333...
- In the same way, no matter how many binary digits you use, the decimal value 1/10 = 0.1 cannot be represented exactly as a binary fraction.
- In base 2, the decimal value 1/10 = 0.1 is the infinitely repeating fraction: 0.0001100110011001 1...
- Stop at any finite number of bits, and you get an approximation!

# A Quick Note of Floating-Point Arithmetic: Example

- Suppose we want to transform a decimal number $n = 4.47$ into its corresponding yet approximated binary fraction using $k = 6$ bits of precision.
- **Step 1:** Conversion of the integer part of $n$ (i.e., $4$) to binary:

    1. 4/2 : Remainder = 0 : Quotient = 2
    2. 2/2 : Remainder = 0 : Quotient = 1
    3. 1/2 : Remainder = 1 : Quotient = 0

So, equivalent binary of integral part of decimal is **100**

# A Quick Note of Floating-Point Arithmetic: Example

- **Step 2:** Conversion of the fractional part of *n* (i.e., *.47*) to binary:

    1. 0.47 * 2 = 0.94, Integral part: 0
    2. 0.94 * 2 = 1.88, Integral part: 1
    3. 0.88 * 2 = 1.76, Integral part: 1
    4. 0.76 * 2 = 1.32, Integral part: 1
    5. 0.32 * 2 = 0.64, Integral part: 0
    6. 0.64 * 2 = 1.28, Integral part: 1

So, equivalent binary of fractional part of decimal is **.011101**

# A Quick Note of Floating-Point Arithmetic: Example

- **Step 3:** Combining the result of Step 1 and 2 to get the *k*-bit ($k = 6$) approximated binary fraction corresponding to the decimal number $n = 4.47$.

$$(4.47)_{10} = 100 + 0.011101 = (100.011101)_2$$

# Operations (except for `complex`)

| Operation | Result |
|---|---|
| `x + y` | sum of *x* and *y* |
| `x - y` | difference of *x* and *y* |
| `x * y` | product of *x* and *y* |
| `x / y` | quotient of *x* and *y* |
| `x // y` | floored quotient of *x* and *y* |
| `x % y` | remainder of `x / y` |
| `-x` | *x* negated |
| `+x` | *x* unchanged |
| `abs(x)` | absolute value or magnitude of *x* |
| `int(x)` | *x* converted to integer |
| `float(x)` | *x* converted to floating point |
| `complex(re, im)` | a complex number with real part *re*, imaginary part *im*. *im* defaults to zero. |
| `c.conjugate()` | conjugate of the complex number *c* |
| `divmod(x, y)` | the pair `(x // y, x % y)` |
| `pow(x, y)` | *x* to the power *y* |
| `x ** y` | *x* to the power *y* |

# Divsion (`/`): Python 2.x vs. Python 3.x

- In Python 2.x, dividing two integers always results in an `int` (C-style).
- In Python 3.x, dividing two integers always returns a `float`.
- This is fine when the result of your integer division is an integer, but it leads to quite different results when the answer is a real number!

```
# Python 2.x
# Division operator (/) always returns an int
print 4/2
2
print 3/2
1
```

```python
# Python 3.x
# Division operator (/) always returns a float
print(4/2)
2.0
print(3/2)
1.5
```

# Integer Division in Python 3.x

- To get C-style integer division in Python 3.x, use the floor division operator `//`:

```
print(3//2)
1
```

# Sequences

# Sequence Types

- Built-in sequences can be either ***immutable*** or ***mutable***.
- ***Immutable*** sequence types are:
    - `str`
    - `bytes`
    - `tuple`
- ***Mutable*** sequence types are:
    - `byte array`
    - `list`

# Operations

The operations in the following table are supported by most sequence types, both ***mutable*** and ***immutable***.

| Operation | Result |
|---|---|
| `x in s` | `True` if an item of *s* is equal to *x*, else `False` |
| `x not in s` | `False` if an item of *s* is equal to *x*, else `True` |
| `s + t` | the concatenation of *s* and *t* |
| `s * n` or `n * s` | equivalent to adding *s* to itself *n* times |
| `s[i]` | *i*th item of *s*, origin 0 |
| `s[i:j]` | slice of *s* from *i* to *j* |
| `s[i:j:k]` | slice of *s* from *i* to *j* with step *k* |
| `len(s)` | length of *s* |
| `min(s)` | smallest item of *s* |
| `max(s)` | largest item of *s* |
| `s.index(x[, i[, j]])` | index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*) |
| `s.count(x)` | total number of occurrences of *x* in *s* |

# Strings: Type `str` (*immutable*)

## String Definition

- You can write *string literals* using either single quotes `'` or double quotes `"`.
- Similarly, multiline strings with line breaks must be enclosed by triple quotes, either `'''` or `"""`.

In [17]:

```
s = 'This is a single-quoted string'
t = "This is a double-quoted string"
u = 'This is a single-quoted string with "double quotes" inside'
v = "This is a double-quoted string with 'single quotes' inside"
w = 'This is a single-quoted string with \'escaped single quotes\' inside'
x = "This is a double-quoted string with \"escaped double quotes\" inside"
```

In [18]:

```
m_s = '''
This is
a multiline string
enclosed by triple single quotes
'''
m_t = """
This is
a multiline string
enclosed by triple double quotes
"""
```

In [19]:

```python
# Count how many lines the string above is made of
# You might expect the result being 3, instead the '\n' character
# right after the opening and closing triple quotes counts as well
len(m_s.split('\n'))
```

Out[19]: 5

# Properties

- Python 3.x strings (`str`) are **immutable** sequences of Unicode **code points**.
- Unicode is a standard mapping between each character of every language to a unique number (**code point**) [to support non-ASCII characters].
- Unicode defines 1,114,112 code points, which are denoted by (hexadecimal) numbers in the range of `U+000000 - U+10FFFF`.
- In Python 2.x, `str` instead refers to a sequence of **bytes** and there is a dedicated type `unicode` for representing Unicode code points.
- You **cannot** modify a string without creating a new one.

In [20]:

```python
s = 'This is a string'
# Try to access the 7-th character of the sequence (index is 0-based)
# and change it to a different character
s[6] = 'z'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-052ef33b03de> in <module>()
      2 # Try to access the 7-th character of the sequence (index is 0-based)
      3 # and change it to a different character
----> 4 s[6] = 'z'

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

In [21]:

```
# This will actually create a new, modified string object
new_s = s.replace('string', 'new string')
print(new_s)
```

```
This is a new string
```

## String Concatenation

- It is often very useful to be able to combine strings into a new string.
- This can be done with the plus sign (`+`), which is the *operator* used to concatenate two (or more) strings into one.
- You can use as many plus signs as you want in composing messages.

In [22]:

```python
a = 'This is the first string.'
b = 'This is the second string.'
c = 'This is the third string.'
# Concatenating them all and interleave each string with a blank character
print(a + ' ' + b + ' ' + c)
```

```
This is the first string. This is the second string. This is the third string.
```

# Quiz Time

Concatenating more than 2 strings using the '`+`' operator doesn't scale well and might be highly inefficient when the number of strings to concatenate becomes larger. **Why?**

# Answer

Because for each concatenation (i.e., for each pair of strings to concatenate) a *new* string object is created (allocated) and all the previous strings have to be first copied into the newly allocated space for result
Suppose you have $n$ strings (therefore $n-1$ concatenations), each string of length $l$: you'll copy $2l$ characters for the first concatenation (i.e., $l$ from the first and $l$ from the second string), plus $3l$ the second concatenation, plus $4l$ the third concatenation, and so on and so forth.
Overall:

$$l * \sum_{i=2}^{n} i = l * \Big[\frac{n(n+1)}{2} - 1\Big],$$

which is, indeed, $O(n^2)$.
[*As of Python 2.4, the CPython implementation avoids creating a new string object when using a += b or a = a + b, but this optimization is both fragile and not portable.*]

# More Efficient String Concatenation

- Use `" ".join([a, b, c])`

In [23]:

```
print(" ".join([a, b, c]))
# alternatively, use a different separator from whitespace (e.g., '\n')
#print("\n".join([a, b, c]))
```

```
This is the first string. This is the second string. This is the third string.
```

## String Formatting

- String templating or formatting is another important topic.
- The number of ways to do so has expanded with the advent of Python 3
- String objects have a `format` method which can be used to substitute formatted arguments into the string, producing a new string.
- More information can be found on Python official [documentation].

In [24]:

```python
# Suppose you have multiple strings that are made of some fixed portion
# as well as some variable portions that all adhere to a specific formatting pattern.
# Let's define the following formatting pattern
template = '{0:.2f} {1:s} are worth US${2:d}'

# In the above template string:
# {0:.2f} means to format the first argument as a floating point number with 2 decimals.
# {1:s} means to format the 2nd argument as a string.
# {2:d} means to format the 3rd argument as an exact integer.

# We perform parameter substitution on the template defined above using the format method
print(template.format(4.5560, 'Argentine Pesos', 1))
```

```
4.56 Argentine Pesos are worth US$1
```

In [25]:

```python
# If the order of the arguments of .format is the same of that expected by template
# you can omit the indices: 0, 1, 2, etc.
template = '{:.2f} {:s} are worth US${:d}'

# We perform parameter substitution on the template defined above using the format method
print(template.format(4.5560, 'Argentine Pesos', 1))
```

```
4.56 Argentine Pesos are worth US$1
```

In [26]:

```python
# Otherwise, you could specify a different order in the template w.r.t. the one of .format
# BE CAREFUL WITH THIS APPROACH!
template = '{2:.2f} {0:s} are worth US${1:d}'

# We perform parameter substitution on the template defined above using the format method
print(template.format(4.5560, 'Argentine Pesos', 1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-150fe06fb6c3> in <module>()
      4
      5 # We perform parameter substitution on the template defined above using the format
 method
----> 6 print(template.format(4.5560, 'Argentine Pesos', 1))

ValueError: Unknown format code 's' for object of type 'float'
```

In [27]:

```python
# Otherwise, you could specify a different order in the template w.r.t. the one of .format
# BE CAREFUL WITH THIS APPROACH!
template = '{2:.2f} {0:s} are worth US${1:d}'

# We perform parameter substitution on the template defined above using the format method
print(template.format('Argentine Pesos', 1, 4.5560))
```

```
4.56 Argentine Pesos are worth US$1
```

# Unicode vs. Byte Strings

# Python 2.x

- In Python 2.x there are 2 distict types of strings:
    - `str` --> refers to sequence of bytes;
    - `unicode` --> refers to sequence of Unicode code points.
- Depending on the **character encoding** used (e.g., UTF-8, ISO 8859-1, etc.) the same code point is possibly mapped to a different sequence of bytes.

# Unicode as an Abstraction of Text

| letter | Unicode Code Point |
|---|---|
| ć | \u0107 |

| Byte Encodings | | | |
|---|---|---|---|
| **letter** | **UTF-8** | **UTF-16** | **Shift-JIS** |
| ć | \xc4\x87 | \x07\x01 | \x85\xc9 |

# From Byte to Unicode String in Python 2.x

- To convert a Python 2.x byte string object (`str`) into its corresponding Unicode object (`unicode`) you need to call the `decode(character_encoding)` method (assuming you know `character_encoding`, e.g., UTF-8)

```
# Assuming this is a UTF-8 encoded Python 2.x str
s = 'This is a UTF-8 byte string' # s has type str
u_s = s.decode("UTF-8") # u_s has type unicode
```

# From Unicode to Byte String in Python 2.x

- Every time you have to serialize out your string you need to transform it into a sequence of bytes!
- To do so, use the `encode(character_encoding)` method.
- **Warning:** Not every Unicode sequence can be encoded by every character encoding! For example, ASCII character encoding can only encode Unicode sequences representing ASCII characters.
- Here is a comprehensive reference to all we have been discussing so far.

# Luckily, We Use Python 3.x!

- Since Python 3.0, Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text.
- Now the type `str` refers to Unicode **not** to bytes!
- There is however a specific type `bytes` to explicitly indicate sequence of bytes.

In [28]:

```python
print('***** From Unicode string to byte string *****')
# This is a Unicode string containing non-ASCII character
s = 'Barça'
# This statement prints the type associated with s
print(type(s))
# We still can convert this Unicode string
# to its UTF-8 bytes representation using the encode method:
s_utf8 = s.encode("utf-8")
print(s_utf8)
print(type(s_utf8))
# If we try to encode our Unicode sequence to ASCII encoding...
s_ascii = s.encode("ascii")
```

```
***** From Unicode string to byte string *****
<class 'str'>
b'Bar\xc3\xa7a'
<class 'bytes'>

---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-28-b1047bf72ce8> in <module>()
     10 print(type(s_utf8))
     11 # If we try to encode our Unicode sequence to ASCII encoding...
---> 12 s_ascii = s.encode("ascii")

UnicodeEncodeError: 'ascii' codec can't encode character '\xe7' in position 3: ordinal not
 in range(128)
```

In [29]:

```python
print('***** From byte string to Unicode string *****')
# Assuming you know the Unicode encoding of a bytes object,
# you can still go back using the decode method:
s_unicode = s_utf8.decode("utf-8")
print(s_unicode)
print(type(s_unicode))
# Again, if we try to decode the byte sequence with a different encoding
# than the one actually used to serialize the Unicode sequence...
s_unicode = s_utf8.decode("ascii")
```

```
***** From byte string to Unicode string *****
Barça
<class 'str'>

---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-29-adbff3796966> in <module>()
      7 # Again, if we try to decode the byte sequence with a different encoding
      8 # than the one actually used to serialize the Unicode sequence...
----> 9 s_unicode = s_utf8.decode("ascii")

UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in ran
ge(128)
```

# Not Everything Needs To Be UTF-8-encoded!

- While it is become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:
    - UTF-16
    - ISO 8859-1 (latin1)
    - Windows-1252 (CP-1252)
    - ...

In [30]:

```python
print(s.encode("utf-16"))
print(s.encode("iso-8859-1"))
print(s.encode("windows-1252"))
```

```
b'\xff\xfeB\x00a\x00r\x00\xe7\x00a\x00'
b'Bar\xe7a'
b'Bar\xe7a'
```

# Bytes: Type `bytes` (*immutable*)

# Sometimes You Just Need Bytes!

- Especially while working with binary files (i.e., files containing sequence of bytes).
- A sequence of bytes is a sequence of integers in the range of `0-255` (only available in Python 3.x).
- You may not want to **decode** those sequence of bytes to Unicode sequence of chars!
- Note however that you can define your own byte literals by prefixing a string with `b`:

```
byte_string = b'This is a byte string'
```

# ByteArray: Type `bytearray` (*mutable*)

# Properties

- This built-in data type corresponds to ***mutable*** `bytes`.
- It is only available in Python 3.x.

# Summary

- Built-in data types:
  - `bool` and `NoneType` (`None`)
  - numeric: `int`, `float`, `complex` (*immutable*)
  - sequences: `str`, `bytes` (*immutable*), `bytearray` (*mutable*)
  - More built-in data types in the next lecture!