An introduction to Python

Builtin types and constructors

Python

To get in touch
with the language

- philosophy
- origin
- backgrounds
- the interpreter

History

Origin

Guido van Rossum

- Conceived in the late 1980s
- Implementation began in 1989
- Focus on readability

Releases

Major releases

- **2.0**: 16 October 2000
- **3.0**: 3 December 2008

Development

Python Enhancement Proposals (PEPs)

- All new features proposed through PEP
- Guido is the **BDFL** (benevolent dictator for life)

A modern programming language

Simplicity

Easy to learn

- Looks like pseudocode
- Easy to read
- Easy to maintain
- But it can be difficult to master...

Flexibility

Multiparadigm

- Imperative
- Object oriented
- Functional
- ...

General purpose

Multidisciplinary

- Science
- Engineering
- Data management
- System administration
- Web
- ...

A modern programming language

Dynamic

No static checks

- No need to compile
- No need to declare variables/functions

Strong typing

Well defined behavior

- The type of every object is completely specified
- *Duck typing*

High-level

High-level

- Suitable for complex programs
- Do not worry about low-level details!

Not a scripting language!

Python is a complete, high-level language.

- It can be used to code very complex programs.
- Not so easy to embed in applications.
- Not an alternative to *bash*.

The zen of python **(PEP 20)**

>>> import this

1. **Beautiful is better than ugly.**
2. **Explicit is better than implicit.**
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. **Readability counts.**
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. **Errors should never pass silently.**
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. **There should be one --and preferably only one--obvious way to do it.**
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!

Performance

The mantra

People are able to code complex algorithms in much less time by using a high-level language like Python. *There can be a performance penalty in the most pure sense of the term.*

```
"The best performance improvement is the transition from the nonworking to the working state."
--John Ousterhout

"Premature optimization is the root of all evil." --Donald Knuth

"You can always optimize it later." -- Unknown
```

Optimization

Libraries

Use fast libraries

- numpy
- scipy
- ...

Tools

Tools

- profilers
- shedskin
- ...

Cython

C + Python = Cython

Usually less than 10% of lines are responsible of more than 90% of the execution time.

Use high-performance languages to code only these very few lines!

Python2 vs Python3

Major differences will be shown

This is what we will learn

Python2

- Version 2.7 is the last 2.X release
- Less disruptive improvements in python 3.X will be backported
- Supported at least until 2020

Not a scripting language!

Python3

- Not backward compatible!
- Version: 3.5
- Devel version: 3.6
- All major libraries are fully compatible
- A neater language
- Easier to learn, more powerful

Interactive vs program file

Interactive

- Prompt (>>>)
- Expression values are directly printed (no need to `print`)

Running a program file

- No prompt
- Expression values must be explicitly printed

The interpreter

An interactive calculator...

```
(python3.5)$ python
Python 3.5.2 (default, Oct 21 2016, 21:46:25)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The interpreter

...

```
>>> print(2 ** 10)
1024
>>> 2 ** 16
65536
>>> quit()
(python3.5)$
```

The interpreter

Running a program file

```
(python3.5)$ cat first-main.py

print(2 ** 10)

2 ** 16

(python3.5)$ python first-main.py

1024

(python3.5)$
```

Python programs

A python program is a text file containing python code. It does not need the `.py` extension.

```
(python3.5)$ cat prog.exe
#!/usr/bin/env python


# an example program
print(2 ** 10)
(python3.5)$
```

UNIX
The interpreter to be used
This is a python comment

This is a python comment

This is a python statement

Running python programs

The python interpreter can be explicitly used to run a python program:

```
(python3.5)$ python prog.exe
1024
(python3.5)$
```

[UNIX] If the file has execution permission, and if the *shebang (#!/usr/bin/env python)* was used, the program can be directly called:

```
(python3.5)$ chmod 755 prog.exe
(python3.5)$ ./prog.exe
1024
```

Python source files

Source encoding

By default, a python source file is written using the *utf-8* encoding (unicode). Any other encoding can be accepted.

The encoding can be defined at the very beginning of the file:

```
# -*- coding: utf-8 -*-
```

Python 2 vs Python 3

Default source encoding

Default encoding is `latin-1`

The `latin-1` encoding includes only the first 127 ASCII characters.

By default you cannot use `à, è, ì, ò, ù` (even in comments!)

Default encoding is `utf-8`

No need to change the encoding!

The `print()` function

The `print()` function can be used to print everything

```
>>> print(2, 3 * 5)
2 15
>>>
```

Normally it prints a newline; this can be avoided

```
>>> print(2, 3 * 5, end='')
2 15>>>
```

Python 2 vs Python 3

print

`print` is a statement:

```
(python2.7)$ python
...
>>> print 2 ** 10
1024
>>>
```

`print()` is a function:

```
(python3.5)$ python
...
>>> print(2 ** 10)
1024
>>>
```

Types

Getting information about live objects

- `int`
- `float`
- `complex`
- `str`
- `bool`
- the None singleton

Integer values

Integer values have arbitrary precision:

```
>>> print(2 ** 1024)

179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216

>>>
```

Strong typing

The `int` type

The type for integer values is `int`:

```
>>> print(type(2))
<class 'int'>
>>>
```

Python 2 vs Python 3

The `int` type

Two integer types: `int`, `long`

```
>>> print type(2 ** 62)
<type 'int'>
>>> print type(2 ** 63)
<type 'long'>
>>> 2 ** 63
9223372036854775808L
```

Only `int`:

```
>>> print(type(2 ** 62))
<class 'int'>
>>> print(type(2 ** 63))
<class 'int'>
>>> 2 ** 63
9223372036854775808
```

Floating point values

Floating point values can be expressed as in other languages:

```
>>> print(2.0)
2.0
>>> print(2.003e10)
20030000000.0
>>> print(2.003e-2)
0.02003
```

Strong typing

The `float` type

The type for floating point values is `float`:

```
>>> print(type(2.0))
<class 'float'>
>>>
```

Floating point values have a limited precision

```
>>> print(2.0 ** -2048)
0.0
>>> print(2.0 ** +2048)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
>>>
```

Mixed int/float arithmetic

Mixed integer/floating point arithmetic always returns floating points (the “bigger” type):

```
>>> 3 * 10.5
31.5
>>> 3.5 - 10
-6.5
>>> 2. ** 3
8.0
>>>
```

Variables

Setting variables is easy as in other languages:

```
>>> i_value = 10

>>> f_value = 2.5 * i_value
```

Variable names are case sensitive.

Nevertheless, these symbolic names are not really variables as in C++ or Fortran!

No type declaration is needed!

Symbolic names do not need to be declared. Moreover, the type of a symbolic name can change during the execution:

```
>>> a = 10
>>> print(type(a))
<class 'int'>
>>> a = 1.5
>>> print(type(a))
<class 'float'>
```

Python variables are **not** variables at all!

- Python variables are simply symbolic names attached to objects.
- There is no type attached to a symbolic name!
- At any time, a symbolic name refers to exactly one object; anyway, that object can change during time.
- Notice that the type of an object never changes through its lifetime, as required by *strong typing*!

PEP 8 - coding guidelines

- Avoid 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye)
- Use only lowercase and underscore for variables

Complex values

A number followed by `j` is imaginary; this can be used to create complex numbers:

```
>>> z = 3.0 + 4.0j

>>> print(z, abs(z))

(3+4j) 5.0

>>> w = 5.0 + 6.0j

>>> print(z * w)

(-9+38j)
```

Strong typing

The `complex` type

The type for complex values is `complex`:

```
>>> print(type(3.0 + 4.0j))
<class 'complex'>
>>>
```

Attributes

In object-oriented programming languages, objects can have named attributes.

For instance, complex numbers have the *real* and *imag* attributes, corresponding to the real and imaginary parts of the number.

Attributes can be accessed using the dot: `z.real` is the real part of the complex object `z`.

```
>>> z = 3.0 + 4.0j

>>> w = 2.5 + 3.5j

>>> z.real  # the real part of z

3.0

>>> w.imag  # the imaginary part of w

3.5

>>>
```

Arithmetic operators

Python has all the common arithmetic operators:

- Addition: **+**
- Subtraction: **-**
- Multiplication: *****
- (True) division: **/**
- **Floor division: //**
- Modulus: **%**
- Exponent: ******

Two division operators

True division

- The standard float division

```
>>> 5.0 / 2
2.5
```

- **`int / int`** returns a **`float`**

```
>>> 5 / 2
2.5
```

Floor division

- **`float // float`** produces a **`float`** without decimal part

```
>>> 5.0 // 2.0
2.0
```

- **`int // int`** returns an **`int`**

```
>>> 5 // 2
2
```

Python 2 vs Python 3

Integer true division

The integer (true) division returns always an `int`:

```
>>> print 5 / 2

2
```

The integer (true) division returns always a `float`:

```
>>> print(5 / 2)

2.5
```

Arithmetic operators

Additional arithmetic operators

Respect to other languages python offers additional arithmetic operators/functions:

- Floor division **`//`**
- Division and modulus **`divmod(...)`**

```
>>> divmod(10, 3)

(3, 1)

>>>
```

Binary assignment operators

Every binary operator has a related binary assignment operator; they are called *in-place operators*:

```
>>> a = 10

>>> a += 3 # -> a = a + 3

>>> a

13
```

The `+=` operator is the increment operator.

Using these in-place operators can increase readability; in the code above, the increment operation on **a** is explicit.

Comparison operators

Python has all the common comparison operators:

- Less than: **<**
- Less than or equal to: **<=**
- Greater than: **>**
- Greater than or equal to: **>=**
- Equal to: **==**
- Not equal to: **!=**

Logical operators

Python offers all the common logical operators:

- `and`
- `or`
- `not`

PEP 8 - coding guidelines

- Avoid tabs!
- Always surround binary operators with a single space on either side
- Always use a single space after comma
- Avoid other spaces

```
>>> a = 10
```

~~`>>> a =   10`~~

```
>>> a = 4 + 5
```

~~`>>> a=4 +5`~~

~~`>>> a = 4   +5`~~

```
>>> a = divmod(10, 3)
```

~~`>>> a = divmod(10,3)`~~

~~`>>> a = divmod( 10, 3 )`~~

~~`>>> a = divmod (10, 3)`~~

Hands-on

Arithmetic operators

1. Open the interactive interpreter and try to use several arithmetic operators
2. Create a program file containing some arithmetic operator

String values

A string can be created using ‘single’ or “double” quotes:

```
>>> p0 = "Hello, "
>>> p1 = ‘world!’
>>> print(p0, p1)
Hello,  world!
>>>
```

Strong typing

The `str` type

The type for strings is `str`:

```
>>> print(type("alpha"))
<class 'str'>
>>>
```

Python 2 vs Python 3

Strings

`str` and `unicode`

In python 2.x the `str` class supports only ascii strings.

The `unicode` class supports unicode strings.

The string base class is *basestring*.

Only `str`

The `str` class is used for both ascii and unicode strings.

There is not *basestring* class.

Triple quotes

Triple quotes can be used for multi-line texts:

```
>>> paragraph = """<p>
...     Lorem ipsum.
... </p>"""
>>> print(paragraph)
<p>
    Lorem ipsum.
</p>
>>>
```

String quotes

`'single', "double"`

Single-line texts

- Cannot span over multiple lines.
- Can contain escape sequences (`'\t'`, `'\n'`, `'\\'` …).

`"""`, `'''`

Multiline texts

- Can span over multiple lines.
- Can contain escape sequences.

r"raw"

Raw strings

- No escape sequences.
- Generally used for regular expressions.

String concatenation

The **+** operator can be used to concatenate strings:

```
>>> a = "Hello, " + 'world' + "!"
>>> print(a)
Hello, world!
```

In addition, adjacent string literals are automatically concatenated

```
>>> a = "Hello, " 'world' '!'
>>> print(a)
Hello, world!
>>>
```

String repetition

The * operator can be used to multiply string by integers; this mean repeating the string:

```
>>> print("a" * 10)
aaaaaaaaaa
>>> print("abc" * 6)
abcabcabcabcabcabc
>>>
```

String escape sequences

Non-raw strings have escape substitution:

```
>>> print("alpha\nbeta\tgamma")
alpha
beta gamma
```

Escape substitution does not happen in raw strings:

```
>>> print(r"alpha\nbeta\tgamma")
Alpha\nbeta\ tgamma
>>>
```

String len

The `len(...)` function returns the string length (the number of items):

```
>>> print(len("alpha"))
5
>>> s = "Hello, world!"
>>> print(len(s))
13
>>>
```

String items

The [ ] operator can be used to access single items.

```
>>> s = "abcd"
>>> s[0]
'a'
>>> s[3]
'd'
>>>
```

Remember:

`s[index]`

- first index is **0**
- last indes is **len(s) - 1**

String indices

Accessing an out-of-range index is an error:

```
>>> s = "abcd"
>>> s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

String negative indices

Negative indices can be used to get items from the right. Indeed, negative indices are automatically increased by the string lengths:

```
>>> s = "abcd"

>>> s[-1]  # -> s[-1 + len(s)] = s[3]

'd'

>>> s[-2]  # -> s[-2 + len(s)] = s[2]

'c'

>>>
```

String slices

Slices can be used to obtain substrings:

```
>>> s = "Hello, world!"
>>> s[2:5]
'llo'
```

Slices can use negative indices:

```
>>> s[-6:-1]
'world'
>>>
```

Remember:

`s[start:stop]`

- **`start`** is always included;
- **`stop`** is always excluded.

Omitting slice indices

Slices indices can be omitted:

```
>>> s = "Hello, world!"
>>> s[:5]
'Hello'
>>> s[5:]
', world!'
>>> s[:]
'Hello, world!'
>>>
```

Remember:

`Missing slice indices`

- **`s[:stop]`** from the beginning to `stop` (excluded);
- **`s[start:]`** from `start` (included) to the end;
- **`s[:]`** a full copy of `s`

Extended slices

Slices accepts also a *stride*, defaulting to 1:

```
>>> s = "Hello, world!"
>>> s[1:-1:2]
'el,wrd'
```

Negative strides are accepted:

```
>>> s[-2::-1]
'dlrow ,olleH'
>>>
```

The in/not in operators

in/not in

The in operator checks if a character or substring is contained in a string:

```
>>> "f" in "abcde"
False
>>> "bcd" in "abcde"
True
>>> "bcx" not in "abcde"
True
```

Methods

In object-oriented programming languages a method is a function attribute. Methods are included in the object's class.

Methods are called using the dot: `a.lower()` calls the lower function ono the string **a**.

```
>>> a = "Hello, world!"
>>> a.lower()
'hello, world!'
>>> a.upper()
'HELLO, WORLD!'
>>> b = "alpha beta gamma"
>>> b.title()
'Alpha Beta Gamma'
>>>
```

String methods

In addition to operators, there are many functions working on strings; for instance:

- `str.upper()`: returns an uppercase copy of the string
- `str.lower()`: returns a lowercase copy of the string
- `str.title()`: returns a titlecased copy of the string
- `str.replace('x', 'y')`: returns a copy of the string where all the `'x'`s are replaced by `'y'`s
- `str.find('x')`: returns the index of the first occurrence of `'x'` in the string
- `str.strip()`: returns a copy of the string without leading/trailing spaces
- `'alpha:beta:gamma'.split(':')`: split a string into words using `':'` as delimiter
- `':'.join(['alpha', 'beta', 'gamma'])`: joins a list of strings

Hands-on

Exercise with strings

1. Create string variables
2. Access string items
3. Access substrings
4. Try using some string method

Converting objects to strings

Any Python object can be converted to a string. Moreover, there are two ways to convert an object to string:

- `str(obj)`: the usual way to print `obj`;
- `repr(obj)`: the object's representation is shown; for builtin types, it's the source code that creates `obj`.

String representation of builtin types

For all builtin types but strings, `str()` and `repr()` return the same string:

```
>>> print(str(10), repr(10))
10 10
>>> print(str(1.3), repr(1.3))
1.3 1.3
>>> print(str(True), repr(True))
True True
>>>
```

String representation of builtin types

The `str` type has instead different str/repr:

```
>>> print(str('alpha'), repr('alpha'))
alpha 'alpha'
>>>
```

The `repr()` function returns the quoted string.

Remember:

For all builtin types, the result of `repr()` is the Python representation of the object; when executed, it creates the same object.

Printing objects

When an object is printed, if it is not a string it is automatically converted to string using `str()`:

```
>>> i = 10
>>> s = "alpha"
>>> print(i, s)
10 alpha
>>>
```

The interpreter

The interactive interpreter shows the result of an expression as `repr()`:

```
>>> i
10
>>> s
'alpha'
>>>
```

Immutable types

Maybe you noticed that string objects cannot be changed. Strings are immutable types.

Believe or not, all the builtin scalar types (`str`, `int`, `float`, `complex`, `bool`, `...`) are immutable! For instance, there is no way to change an integer object.

Strings are immutable

It is not possible to change a string item:

```
>>> s = "Hello, world!"
>>> s[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Strings are immutable

None of the string methods can be used to change the string.

For instance, `str.replace(...)` returns another string, but the original one is left unchanged:

```
>>> s1 = s.replace('o', '0')
>>> s1
'Hell0, w0rld!'
>>> s
'Hello, world!'
>>>
```

Strings are immutable

How about increment operator? We can use it to concatenate a string in-place:

```
>>> s = "Hello, "
>>> s += "world!"
>>> s
'Hello, world!'
```

Is the `str` object really changed? The answer is no.

The id( ) function

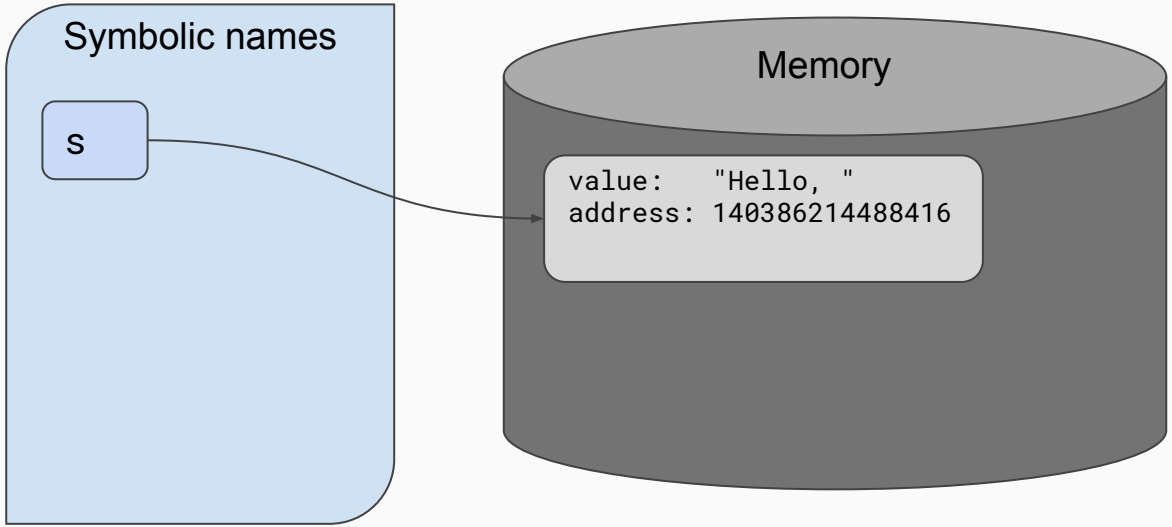
The id() function returns the unique id of a python object.

```
>>> s = "Hello, "

>>> id(s)

140386214488416
```

The id( ) function

The id() function returns the unique id of a python object.

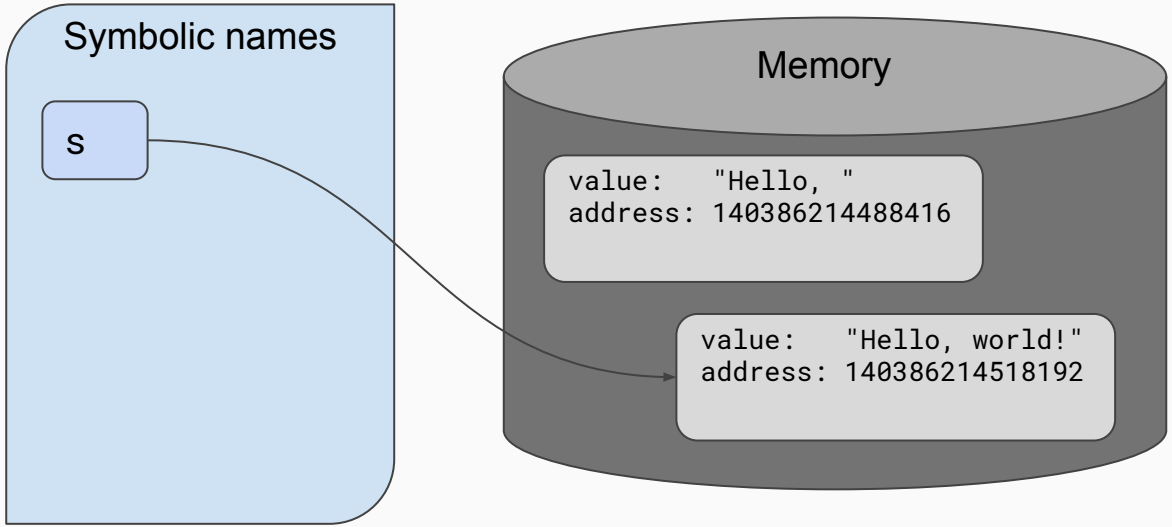
```
>>> s = "Hello, "
>>> id(s)
140386214488416
```



The id( ) function

The id() function returns the unique id of a python object.

```
>>> s = "Hello, "
>>> id(s)
140386214488416
>>> s += "world!"
>>> s
'Hello, world!'
>>> id(s)
140386214518192
```



The in-place operators

For all the builtin types seen so far, the *in-place* operators (`+=`, `-=`, `*=`, …) are not *in-place* at all. They are completely equivalent to the long version

- a += b
- a= a + b

Anyway, always use *in-place* operators when the operation is *semantically in-place*.

Python data model

Object reference

A symbolic name refers to an object.

The symbolic name can be moved to another object.

The object itself can be mutable or immutable. All the built-in types seen so far are immutable.

All numeric types are immutable

Numeric types are immutable too:

```
>>> a = 10
>>> id(a)
140386213813280
>>> a += 3
>>> id(a)
140386213813376
>>>
```

Comments

Python considers a comment the source code from the # symbol to the end of the line:

```
>>> size *= factor  # apply the scaling factor

>>>
```

PEP 8 - coding guidelines

- Use inline comments sparingly.
- Avoid comments stating the obvious:

```
>>> x += 1  # increase x by 1
```

- Use only useful comments:

```
>>> x += 1  # compensate for border
```

Hands-on

Put some of the string exercises in a program file and run it.

None

The None value can be used for variables whose value is currently undefined:

```
>>> a = None
>>> print(a)
None
```

None is an object, not a type; it's a *singleton*.

Bool

Boolean objects can be used to store a truth value. Two bool literals are available: `True` and `False`.

```
>>> t = True
>>> print(t)
True
>>> print(1 > 3)
False
```

Strong typing

The `bool` type

The type for boolean objects is `bool`:

```
>>> print(type(True))
<class 'bool'>
>>>
```

Implicit bool conversion

All the builtin types have a boolean value:

- `int`: 0 is `False`, other values are `True`
- `float`: 0.0 is `False`, other values are `True`
- `str`: the empty string is `False`, non-empty strings are `True`
- None: always `False`

Containers

One of the most powerful features

- tuple ( )
- list [ ]
- set { }, frozenset
- dict { }
- collections
  - deque
  - OrderedDict
  - Counter
  - defaultdict

Tuples ( )

Tuples are immutable sequences of items of arbitrary type.

```
>>> t = (1, "alpha", 3.1)
>>> t
(1, 'alpha', 3.1)
>>> type(t)
<class 'tuple'>
>>>
```

Tuple construction

Parentheses are not really needed to create a tuple; the comma is mandatory instead:

```
>>> a = 1, 2, 3
>>> print(a)
(1, 2, 3)
```

Tuple construction

Comma is needed also if the tuple has a single item:

```
>>> a = (1)
>>> print(a, type(a))
1 <class 'int'>
>>> a = (1, )
>>> print(a, type(a))
(1,) <class 'tuple'>
>>>
```

This is an integer with useless parentheses

This is a tuple; parentheses are not needed

Tuple construction

Parentheses are needed only to define the empty tuple:

```
>>> a = ()
>>> print(a, type(a))
() <class 'tuple'>
>>>
```

Tuple concatenation/repetition

As strings, tuples can be concatenated using the operator +:

```
>>> t = (1, 2, 3)
>>> t + (4, 5)
(1, 2, 3, 4, 5)
```

As strings, tuples can be multiplied by integers, meaning repetition:

```
>>> (1, 2) * 3
(1, 2, 1, 2, 1, 2)
```

Tuple item access

Item access and slicing follow the same syntax as strings (indexing returns the item, slicing returns a sub-tuple):

```
>>> t[-1]
3.1
>>> t[1:]
('alpha', 3.1)
>>> len(t)
3
>>>
```

Item access/slicing

Even it the syntax is (almost) the same, item access and slicing are **not** the same:

```
>>> a = (0, 1, 2, 3)
>>> a[1]
1
>>> a[1:2]
(1,)
>>>
```

This is the tuple item with index 1

This is the sub-tuple containing only the item with index 1

Tuple item assignment

Item assignment is forbidden (tuples are immutable!)

```
>>> t[1] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Tuple methods

The `tuple` class has some methods:

- `t.count(1)` returns the number of occurrences of the object 1 in the tuple `t`
- `t.index(1)` returns the index of the first occurrence of the object 1 in the tuple

Tuple assignment

Tuple of variables on the left-hand of an assignment

When the right-hand of an assignment is a tuple, the left-hand can be a tuple of variables. For instance, we know that `divmod(...)` returns a 2-tuple:

```
>>> div, mod = divmod(10, 3)

>>> div

3

>>> mod

1
```

Tuple assignment

Variable swap

Tuple assignment can be used to swap variables:

```
>>> a = 10
>>> b = 1
>>> a, b = b, a
>>> print(a, b)
1 10
```

Item lookup

The `in`/`not in` operators

The `in` operator can be used to check if a container has a given item:

```
>>> t = (1, 2, 3)
>>> 2 in t
True
>>> 4 in t
False
```

The `not in` operator is also available.

Hands-on

Exercise with tuples

1. Tuple circular shift: given an arbitrary tuple, create a new one by moving the last item to the beginning

Lists [ ]

Lists are mutable tuples

```
>>> lst = [1, 'abc', 5.5]
>>> lst[-1]
5.5
>>> lst[1:]
['abc', 5.5]
>>>
```

List item assignment

Item assignment is supported (lists are mutable!)

```
>>> lst[1] = 10
>>> lst
[1, 10, 5.5]
>>>
```

List item deletion

Item can also be deleted:

```
>>> del lst[1]
>>> lst
[1, 5.5]
>>>
```

The append method

The `list.append(item)` method can be used to add items to the end of the list:

```
>>> lst
[1, 5.5]
>>> lst.append('xyz')
>>> lst
[1, 5.5, 'xyz']
```

The insert method

The `list.insert(index, item)` method can be used to add items at arbitrary positions:

```
>>> lst
[1, 5.5, 'xyz']
>>> lst.insert(2, 44)
>>> lst
[1, 5.5, 44, 'xyz']
>>>
```

The extend method

The `list.extend(sequence)` method can be used to add multiple items coming from sequence:

```
>>> lst
[1, 5.5, 44, 'xyz']
>>> lst.extend([2, 5, 7])
>>> lst
[1, 5.5, 44, 'xyz', 2, 5, 7]
>>>
```

append and extend

Do not be confused about `append()` and `extend()`:

Append inserts an item (it can be a sequence)

```
>>> l2 = [0, 1]
>>> l2.append([2, 3, 4])
>>> l2
[0, 1, [2, 3, 4]]
>>>
```

Extend needs a sequence argument

```
>>> l2 = [0, 1]
>>> l2.extend([2, 3, 4])
>>> l2
[0, 1, 2, 3, 4]
>>>
```

List slicing assignment

Items can be assigned also with slicing:

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst[2:] = [5, 4, 3, 2, 1]
>>> lst
['a', 'b', 5, 4, 3, 2, 1]
>>>
```

List slicing deletion

Slicing can be used also to delete parts of a string:

```
>>> lst = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del lst[2:-1]
>>> lst
['a', 'b', 'f']
>>>
```

List as a stack

A list can be used as a stack (a Last-In-First-Out container):

```
>>> lst = []
>>> lst.append(2)
>>> lst.append(5)
>>> lst.pop()
5
>>> lst.pop()
2
```

The push operation

The pop operation

The `list.pop(index=-1)` method

The pop method accepts as argument the index (it defaults to -1):

```
>>> lst = [1, 2, 3]
>>> lst.pop(1)
2
>>> lst
[1, 3]
>>>
```

Other list methods

Some other methods are available:

- `lst.sort()` sorts the list *in-place*
- `lst.reverse()` reverses the list *in-place*
- `lst.copy()` returns a copy of the list (as `lst[:]`)
- `lst.clear()` clears the list (as `del lst[:]`)

Mutable types and *in-place* operators

Mutable types allow optimized *in-place* operators.

Remember that `obj0 += obj1` has the same effect as `obj0 = obj0 + obj1`; if `obj0` is a mutable object, instead of creating a new object, the original object is changed in place.

In-place operators

Immutable types

```
>>> tpl = (1, 2, 3)

>>> id(tpl)

140106958666560

>>> tpl += (4, 5)

>>> id(tpl)

140106959260840
```

Mutable types

```
>>> lst = [1, 2, 3]

>>> id(lst)

140106958659080

>>> lst += [4, 5]

>>> id(lst)

140106958659080
```

Hands-on

Exercise with tuples

1. List circular shift: move the last item of an arbitrary list to the beginning *in-place*

Sequences

Lists, tuples and strings are all *sequences*. Sequences consist of items that can be accessed in a given direction (from left to right).

A string is the sequence of its characters.

Functions operating on sequences

Iteration over sequences is pervasive in python. Many functions work on sequences.

For instance, list and tuple constructors accept a sequence as argument; this allow to construct a new container from an existing one:

```
>>> s = "abc"

>>> tpl = tuple(s)

>>> tpl

('a', 'b', 'c')

>>> list(tpl)

['a', 'b', 'c']
```

Sequences that are not containers

All containers are sequences; nevertheless, a sequence is not necessarily a container.

Sequences can be simply *a* recipe to generate all the items, one after the other, from left to right.

The range function generates a sequence of natural numbers. This sequence can be used for instance to create a container (but we will see other uses later):

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(3, 17, 2))
(3, 5, 7, 9, 11, 13, 15)
```

Functions generating sequence items as `range()` are called *generators*.

Python 2 vs Python 3

The range function

range() returns a list

- The `xrange()` is a generator

range() is a generator

Question

Why do we need tuples? List are mutable sequence; they support all tuple’s operations, plus some other.

Set { }

A set is an unordered collection of unique items of arbitrary types.

Sets are particularly efficient for item lookup, insertion and deletion.

```
>>> {1, 'a', 3, 5.3}
{1, 3, 5.3, 'a'}
>>>
```

The original order is not preserved!

Sets do not support indexing

Sets do not support indexing

```
>>> st = {1, 3, 1}
>>> st[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>>
```

This is because sets are unordered.

Empty sets

Empty sets cannot be created using the `{}` syntax: indeed `{}` is an empty `dict`! Use `set()` to create an empty set:

```
>>> obj = {}
>>> type(obj)
<class 'dict'>
>>> obj = set()
>>> type(obj)
<class 'set'>
>>>
```

Set operations

Sets operations are supported:

- `s0.issubset(s1)`: check if the set `s0` is a subset of `s1` (`s0 <= s1`)
- `s0.issuperset(s1)`: check if the set `s0` is a superset of `s1` (`s0 >= s1`)
- `s0.union(s1)`: return the union of `s0` and `s1` (`s0 | s1`)
- `s0.intersection(s1)`: return the intersection of `s0` and `s1` (`s0 & s1`)
- `s0.difference(s1)`: return the difference of `s0` and `s1` (`s0 - s1`)
- `s0.symmetric_difference(s1)`: return the set of elements that are in `s0` or in `s1` but not in both (`s0 ^ s1`)
- `s0.copy()`: returns a copy of `s0`
- `s0 < s1`: check if `s0` is a proper subset of `s1`
- `s0 > s1`: check if `s0` is a proper superset of `s1`

Set is mutable

These methods/operators change the set *in-place*:

- `s0.update(s1)`: update the set `s0` adding elements from `s1` (`s0 |= s1`)
- `s0.intersection_update(s1)`: update the set `s0` keeping only elements in both `s0` and `s1` (`s0 &= s1`)
- `s0.difference_update(s1, s2, ...)`: update the set `s0` removing elements in all others (`s0 -= s1`)
-
- `s0.symmetric_difference_update(s1)`: update the set `s0` keeping only elements in `s0` ondr `s1` but not in both (`s0 ^= s1`)
- `s0.add(item)`: add an item to set `s0`
- `s0.remove(item)`: remove an item from the set `s0`
- `s0.discard(item)`: remove an item from set `s0`, if present
- `s0.pop()`: remove and returns an item
- `s0.clear()`: clears the set

Set items must be *hashable*

Hashable objects

An object is *hashable* if it has a *hash* value which never changes through the object's lifetime.

All immutable types are hashable; by default, mutable types are not hashable.

Hashability is required for lookup efficiency.

Lists and sets, which are unhashable, cannot be used as set values.

Frozenset

A *frozenset* is an immutable set. It supports only methods that do not change the *frozenset* in-place.

```
>>> file_formats = frozenset({'raw', 'text', 'json'})
>>> 'raw' in file_formats
True
>>> file_formats.add('toml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Frozenset

Frozensets are hashable:

```
>>> s0 = {1, 2}
>>> s0.add({4, 5})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> s0.add(frozenset({4, 5}))
>>>
```

Dict { }

A dict is an unordered mapping of unique keys onto values. Both keys and values can have arbitrary types, but keys must be hashable objects. As sets, dicts are very efficient for key lookup.

```
>>> mendeleev = {'H': 1, 'He': 2, 'C': 6, 'O': 8}

>>> mendeleev['N'] = 7

>>> mendeleev

{'O': 8, 'He': 2, 'H': 1, 'C': 6, 'N': 7}

>>> mendeleev['He']

2
```

The original order is not preserved!

Dict key access

Accessing a missing key raises a `KeyError`:

```
>>> mendeleev
{'O': 8, 'He': 2, 'H': 1, 'C': 6, 'N': 7}
>>> mendeleev['Au']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Au'
>>>
```

Dict methods

The following methods do not change the dict object:

- `dct.keys()`: return a sequence of the keys in `dct`
- `dct.values()`: return a sequence of the values in `dct`
- `dct.items()`: return a sequence of the items in `dct` (items are `(key, value)` tuples)
- `dct.get(key, default=None)`: return the value for `key` if it is in `dct`, else `default`
- `dct.copy()`: return a copy of `dct`
- `dict.fromkeys(iterable, value=None)`: return a new dict with keys from `iterable`, and values all equal to `value`

Python 2 vs Python 3

Dict methods

keys(), values(), items() return lists

- `dct.keys()->[]`
- `dct.iterkeys()`, `dct.itervalues`, `dct.iteritems()` return sequences

keys(), values(), items() generators

Dicts are mutable

These methods change the dict *in-place*:

- `dct.setdefault(key, default=None)`: if `key` is in `dct`, return the corresponding value; otherwise, insert `key` in `dct` with value `default`, and return `default`
- `dct.clear()`: clear the dictionary
- dct.pop(key[, d]): if `key` it it is in `dct`, return the corresponding value and remove it; otherwise, return d or raise a `KeyError`
- dct.popitem(): return an item

Hands-on

Use a dict to store book titles and authors.

Exercise with some of the dict methods.

Other standard containers

The python standard module `collections` contains other useful containers:

- collections.deque: a double ended queue
- collections.OrderedDict: a dict that maintains the key insertion order
- collections.Counter: a dict that counts items (an alternative to *multiset* or *bag*)
- collections.defaultdict: a dict returning a default value for missing keys

deque

A deque is a double ended queue; the `maxlen` argument can set a maximum capacity:

```
>>> from collections import deque
>>> dq = deque([1, 2, 3, 4, 5], maxlen=3)
>>> dq
deque([3, 4, 5], maxlen=3)
```

deque

```
>>> dq.append(6)
>>> dq
deque([4, 5, 6], maxlen=3)
>>> dq.appendleft(0)
>>> dq
deque([0, 4, 5], maxlen=3)
```

The leftmost item 3 has been removed

The rightmost item 6 has been removed

OrderedDict

An `OrderedDict` maintains the original key order:

```
>>> from collections import OrderedDict
>>> dct = OrderedDict()
>>> dct['b'] = 10
>>> dct['a'] = 5
>>> dct['c'] = 15
>>> dct
OrderedDict([('b', 10), ('a', 5), ('c', 15)])
```

Counter

A `Counter` counts items:

```
>>> from collections import Counter
>>> cnt = Counter("abracadabra")
>>> cnt
Counter({'a': 5, 'b': 2, 'r': 2, 'd': 1, 'c': 1})
>>>
```

Counter

A `Counter` is a valid alternative to the `multiset` or `bag` container. It also accept negative counts (this is not possible with a `multiset`).

```
>>> c

Counter({'r': 2, 'b': 2, 'c': 1, 'd': 1, 'a': -5})

>>>
```

defaultdict

A `defaultdict` does not raise an error when accessing a missing key; it calls a `default_factory()` function instead:

- `collections.defaultdict(default_factory)`

This is a first example of passing a function (the `default_factory`) as an argument to another function (the `defaultdict` constructor).

Containers and implicit bool conversion

When used as a conditional expression, a container evaluates to `False` if it is empty, otherwise to `True`.