

An introduction to Python

I/O

Control flow statements

String formatting



Input/Output

A minimal reference for
reading/writing files

- Writing files
- Reading files

The open() function

The open function can be used to open a file; it accepts the filename and the mode:

```
>>> with open("a.txt", "w") as file:  
...     file.write("Hello, world!\n")  
...  
14  
>>>
```

The `file.write(...)` function
returns the number of written bytes

The open() function


```
>>> with open("a.txt", "r") as file:
```

```
...     print(file.read())
```

```
...
```

```
Hello, world!
```

```
>>>
```



The `file.read(...)` reads the file content, `'\n'` included; then printed, there is an additional newline.

Control flow

Python support for imperative programming

- `if/elif/else`
- `for, while`
 - `break`
 - `continue`
- `pass`
- `functions`

Control flow statements

As other imperative programming languages Python has control flow statements:

- Conditional: `if/elif/else`
- Loops: `for, while`
- Functions: `def, lambda`

Indentation

One of the most distinctive Python features is *syntactic indentation*.

Since indenting code is always a good idea, Python enforces indentation: code blocks are only delimited by indentation. No braces or *end do* statements are available!

This strongly increases code readability.

PEP 8 - coding guidelines

- Always indent 4 spaces
- Never use tabs

Conditional: if/elif/else

```
>>> if a > b + c:  
...     print(a)  
... elif b > a + c:  
...     print(b)  
... elif c > a + b:  
...     print(c)  
... else:  
...     print(0)
```

Loop: for

The for statement is used to iterate over a sequence:

```
>>> for item in (1, 2, 3):  
...     print(item)  
...  
1  
2  
3  
  
>>>
```

Loop: for

The sequence is not necessarily a container:

```
>>> for item in range(3):  
...     print(item)  
...  
0  
1  
2
```

Loop: while

The while loop iterates while the condition is true:

```
>>> i = 0
>>> while i < 2:
...     print(i)
...     i += 1
...
0
1
```

Loop: break

Both for and while loops can be prematurely interrupted with the **break** directive:

```
>>> for item in range(1000000):  
...     if item > 1:  
...         break  
...     print(item)  
...  
0  
1
```

Loop: else

Quite surprisingly, loops can have an `else` clause, which is executed only if the loops ends normally, without encountering a `break`.

```
>>> for i in range(3):  
...     if i > 100:  
...         break  
... else:  
...     print("done.")  
...  
done.
```

Loop: else

The loop else clause can be useful for instance to implement a search loop:

```
>>> for item in lst:
...     if item == 10:
...         found = True
...         break
...     else:
...         found = False
... 
```

Loop: continue

The `continue` directive interrupts the current iteration; the loop immediately pass to the following iteration:

```
>>> for item in (2, 4, 6, 7, 8, 10):  
...     if item % 2 == 0:  
...         continue  
...     print(item)  
...  
7  
  
>>>
```


Iteration on containers

Use always the for loop to iterate on containers:

```
>>> for item in [1, 2, 3]:  
...     print(item)  
...  
1  
2  
3  
  
>>>
```

Avoid Fortran/C style iteration!

Using indices to iterate on lists/tuple is an **antipattern**!

```
>>> lst = [1, 3]
>>> for i in range(len(lst)):
...     print(lst[i])
...
1
3
>>>
```

Use always `for item in container`

Notice also that iterating on indices does not work with sets: indeed sets do not support indexing. Anyway, standard iteration with the `for` loop is possible (even if the iteration order is unpredictable):

```
>>> st = {1, 3, 1}
>>> for item in st:
...     print(item)
...
3
1
>>>
```

Iteration on dicts

When iterating on a dict, keys are returned:

```
>>> dct = {'a': 1, 'b': 2}
```

```
>>> for key in dct:
```

```
...     print(key)
```

```
...
```

```
b
```

```
a
```

```
>>>
```

Iteration on dict keys

Nevertheless, it is possible to explicitly iterate on keys:

```
>>> for key in dct.keys():  
...     print(key)  
...  
b  
a  
>>>
```

Iteration on dict values

...or on values:

```
>>> for value in dct.values():  
...     print(value)  
...  
2  
1  
>>>
```

Iteration on dict items

...or on items:

```
>>> for item in dct.items():  
...     print(item)  
...  
( 'b', 2)  
( 'a', 1)  
>>>
```

Unpacking tuples on iteration

When the iteration item is a sequence, it can be unpacked:

```
>>> for key, value in dct.items():  
...     print(key, "=", value)  
...  
b = 2  
a = 1  
>>>
```


The pass statement

The indented body of a control flow statement cannot be empty. The pass statement does nothing, but can be used as a placeholder where some statement is expected:

```
>>> if a > b:
...     pass
... else:
...     print(10)
...
```

Hands-on

Given the book/author dict, for each author print the name and the number of written books.

Hands-on

Print all the prime divisors of 2009. Do not worry about performances.

Functions

Functions can be defined using the def keyword:

```
>>> def add(a, b):  
...     return a + b  
...  
>>> add(2, 4)  
6  
>>>
```

No argument type declaration

Notice that arguments are not declared. This function will work on every two objects *a*, *b* supporting “addition” (*duck typing*):

```
>>> add(3.4, 5.5)
```

```
8.9
```

```
>>> add("Py", "thon")
```

```
'Python'
```

```
>>> add([1, 2], [2, 3])
```

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

```
>>>
```

No argument type declaration

An error is raised only if the addition is not supported by operands:

```
>>> add("alpha", 3.4)
```

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

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

```
  File "<stdin>", line 2, in sum
```

```
TypeError: Can't convert 'float' object to str implicitly
```

```
>>>
```

Recursive functions

A function can call itself (directly or indirectly):

```
>>> def fibonacci(i):  
...     if i < 2:  
...         return i  
...     else:  
...         return fibonacci(i - 2) + fibonacci(i - 1)  
...  
>>>
```



**A very inefficient
implementation!**

Recursive functions

```
>>> for i in range(10):  
...     print(i, fibonacci(i))  
...  
0 0  
1 1  
...  
8 21  
9 34
```


Beware of stack limitation

Due to the stack-based function call, do not use recursive functions when the recursion length is not very limited.

By default, the recursion limit is set to 1000. It can be changed, but it's not a good idea: remember that **for each recursive algorithm an iterative alternative exists**.

Return value

Every function has a default value; if not specified, it is None:

```
>>> def fun():
```

```
...     print("Hi")
```

```
...
```

fun() -> None

```
>>> def fun2():
```

```
...     print("Hi")
```

```
...     return
```

```
...
```

fun2() -> None

Function return type

Function return type is not declared. A function can contain several return statements, each of them returning objects of different types.

Prefer writing functions with homogeneous return type.

Hands-on

Write a function returning all the prime divisors of an arbitrary positive integer.

Function call

When a function is called, arguments can be passed

- **positionally**, or
- **by name**.

Passing positional arguments

```
>>> def count(sequence, value):  
...     n = 0  
...     for item in sequence:  
...         if item == value:  
...             n += 1  
...     return n  
...  
>>>
```

Passing positional arguments

We already know how to pass arguments positionally:

```
>>> count([1, 2, 1, 1, 5, 1], 1)
```

```
4
```

```
>>>
```

This requires knowing the exact position and meaning of each argument.

Passing keyword arguments

It's easier to remember meaningful names than positions:

```
>>> count(sequence=[1, 2, 1, 1, 5, 1], value=1)
```

```
4
```

```
>>> count(value=1, sequence=[1, 2, 1, 1, 5, 1])
```

```
4
```

```
>>>
```


Default argument values

Functions can have arguments with a default value.

```
>>> def get(dct, key, default=None):  
...     if key in dct:  
...         return dct[key]  
...     return default  
...  
>>>
```

Default argument values

```
>>> table = {'a': 1, 'b': 2}
```

```
>>> get(table, 'a')
```

```
1
```

```
>>> get(dct=table, default=10, key='b')
```

```
2
```

```
>>> get(dct=table, default=10, key='c')
```

```
10
```

```
>>>
```

Pitfall - mutable default values

Never use mutable objects as default values in functions!

Indeed, default value binding occurs when the function is defined, not when it is called.

```
>>> def push(value, lst=[]):  
...     lst.append(value)  
...     return lst  
...  
  
>>>  
  
>>> push(10)  
  
[10]  
  
>>> push(20)  
  
[10, 20]  
  
>>>
```

Pitfall - mutable default values

Never use mutable objects as default values in functions!

Indeed, default value binding occurs when the function is defined, not when it is called.

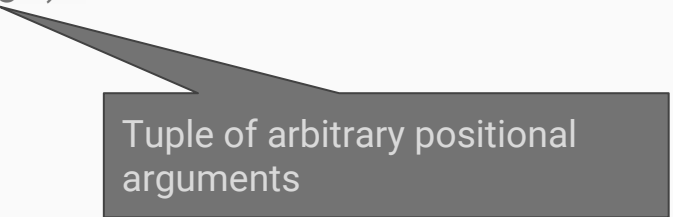
Use None as default value instead!

```
>>> def push(value, lst=None):  
...     if lst is None:  
...         lst = []  
...     lst.append(value)  
...     return lst  
...  
  
>>> push(10)  
  
[10]  
  
>>> push(20)  
  
[20]
```

Packing arbitrary positional arguments

Functions can have arbitrary positional arguments; they are all packed in a tuple. The following function can be called with two or more arguments:

```
>>> def addn(first, second, *args):  
...     s = first + second  
...     for item in args:  
...         s += item  
...     return s
```



Tuple of arbitrary positional arguments

Arbitrary positional arguments

```
>>> addn(1, 2)
```

```
3
```

```
>>> addn(1, 2, 3, 4, 5)
```

```
15
```

```
>>> addn(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
45
```

```
>>>
```

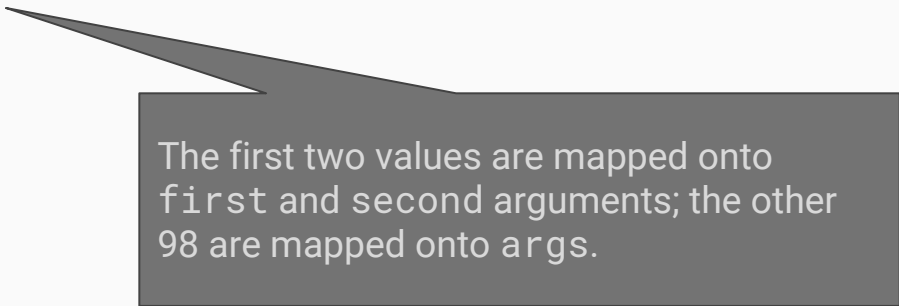
Unpacking positional arguments

What if we want to use `addn` to sum all the positive integer less than 100? Writing the function call by hand is not an option. Notice that we have a sequence containing all the necessary arguments, `range(100)`.

```
>>> addn(*range(100))
```

```
4950
```

```
>>>
```



The first two values are mapped onto `first` and `second` arguments; the other 98 are mapped onto `args`.

Unpacking arguments

Without the *, the previous function call would fail, since addn would receive a single argument which is the sequence, but it requires at least two arguments:

```
>>> addn(range(100))
```

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

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

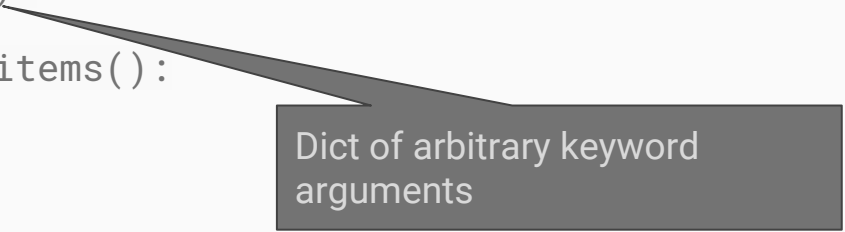
```
TypeError: addn() missing 1 required positional argument: 'second'
```

```
>>>
```


Packing arbitrary keyword arguments

Functions can have arbitrary keyword arguments; they are all packed in a dict.

```
>>> def update_dict(dct, **kwargs):  
...     for key, value in kwargs.items():  
...         dct[key] = value  
...  
>>>
```



Dict of arbitrary keyword arguments

Arbitrary keyword arguments

```
>>> data = {'a': 1, 'b': 2}

>>> update_dict(data, x=10, y=20, z=30)

>>> data

{'b': 2, 'y': 20, 'x': 10, 'z': 30, 'a': 1}

>>>
```

Unpacking keyword arguments

Keyword arguments can be unpacked. For instance:

```
>>> d1 = {'a': 1, 'b': 2}
>>> d2 = {'x': 10, 'w': 20}
>>> update_dict(d1, **d2)
>>> d1
{'w': 20, 'b': 2, 'x': 10, 'a': 1}
>>>
```

Function arguments unpacking

Function arguments unpacking obviously works also with functions that do not accept arbitrary arguments:

```
>>> def sqdistance(x, y, z):  
...     return x ** 2 + y ** 2 + z ** 2  
...  
>>> point = (3, 4, 5)  
>>> sqdistance(*point)  
50
```

Function arguments unpacking

```
>>> point = {'x': 3, 'y': 4, 'z': 5}
```

```
>>> sqdistance(**point)
```

```
50
```

```
>>>
```

Keyword-only arguments

Keyword-only arguments can be passed only by name, not positionally. All arguments between `*args` (or `*`) and `**kwargs` are keyword-only:

```
>>> def get(dct, key, *, default=None):  
...     if key in dct:  
...         return dct[key]  
...     else:  
...         return default
```

Arbitrary keyword arguments

```
>>> dct = {'a': 1, 'b': 2}
```

```
>>> print(get(dct, 'c', default=10))
```

```
10
```

```
>>> print(get(dct, 'c', 10))
```

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

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

```
TypeError: get() takes 2 positional arguments but 3 were given
```

```
>>>
```

Python 2 vs Python 3

Keyword-only arguments

No keyword-only arguments.

Keyword-only arguments are supported.

General function signature

Putting all together, the most general form of a function is:

positional or
keyword

positional
with default

arbitrary
positional

keyword-only
without default
(mandatory)

keyword-only
with default

arbitrary
keyword

```
>>> def function(a, b=10, *args, c, d=20, **kwargs):  
...     print(a, b, c, d, args, kwargs)  
...  
>>>
```

Introspection

Obtaining information
about live objects

- type, id
- doc strings
- inspect

Type introspection

In computing, type introspection is the ability of a program to examine the type or properties of an object at runtime (*wikipedia*).

Introspection can be used for

- learning
- debugging
- write some algorithms

Functions returning information about objects

We already saw the `type()` and `id()` functions: they both return information about live objects (the *type* and the *unique id* respectively).

```
>>> dct = {'a': 1, 'b': 2}
```

```
>>> type(dct)
```

```
<class 'dict'>
```

```
>>> id(dct)
```

```
140478574505096
```

```
>>>
```

The dir() function

The dir() function shows the attributes and methods of an object/type:

```
>>> dir(dict)

['__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
 'setdefault', 'update', 'values']

>>>
```

The help() function

The help() function shows the documentation:

```
>>> help(dict)
```

Help on dict object:

```
class dict(object)
| dict() -> new empty dictionary
| dict(mapping) -> new dictionary initialized from a mapping object's
| (key, value) pairs
| dict(iterable) -> new dictionary initialized as if via:
|     d = {}
|     for k, v in iterable:
|         d[k] = v
| dict(**kwargs) -> new dictionary initialized with the name=value pairs
| in the keyword argument list. For example: dict(one=1, two=2)
|
| Methods defined here:
|
| __contains__(self, key, /)
|     True if D has a key k, else False.
|
| __delitem__(self, key, /)
|     Delete self[key].
|
| __eq__(self, value, /)
|     Return self==value.
|
| __ge__(self, value, /)
|     Return self>=value.
|
| __getattr__(self, name, /)
|     Return getattr(self, name).
|
| __getitem__(...)
|     x.__getitem__(y) <==> x[y]
|
| __gt__(self, value, /)
|     Return self>value.
|
| __init__(self, /, *args, **kwargs)
|     Initialize self. See help(type(self)) for accurate signature.
|
```

The `help()` function and *docstrings*

How does this help function work?

In Python, documentation about types, function, modules etc... is part of the objects themselves.

Functions have a *docstring*, which is a python string containing a (brief) documentation about the function itself. The docstring is contained in the function attribute `__doc__`; all functions have this attribute, which is `None` by default. Builtin functions and methods have a brief but useful docstring:

```
>>> print(dict.get.__doc__)
```

```
D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
```

```
>>>
```

The `help()` function and *docstrings*

So, the `help()` function uses introspection to get

- the object type (`type(dct) -> dict`)
- the type docstring (`dict.__doc__`)
- the type content (`dir(dict) -> [...]`)
 - for each method, its docstring (`dict.get.__doc__`)

Adding custom docstrings

We can also add docstrings to our functions (or types, or modules...). The function docstring is a python (typically multi-line) string just after the def statement:

```
>>> def addn(first, second, *args):
...     """Returns the sum of two or more arguments:
...         >>> addn(1, 2)
...         3
...         >>> addn(*range(10))
...         45
...     """
...     s = first + second
...     for item in args:
...         s += item
...     return s
```

Using `help()` on custom functions

Now we can use `help()` to inspect our `addn` function:

```
>>> help(addn)
```

```
Help on function addn in module __main__:
```

```
addn(first, second, *args)
```

```
    Returns the sum of two or more arguments:
```

```
>>> addn(1, 2)
```

```
3
```

```
>>> addn(*range(10))
```

```
45
```

```
(END)
```

The inspect module

An entire module is dedicated to introspection.

The inspect module contains functions to get information about live objects. As an example, the inspect.getsource function returns the source code of a function loaded from some module:

```
>>> print(getsource(getsource)[:200] + '...')
```

```
def getsource(object):
```

```
    """Return the text of the source code for an object.
```

```
    The argument may be a module, class, method, function, traceback, frame,  
    or code object. The source code is re...
```

The doctest module

The doctest uses introspection to check if examples contained in docstrings are correct:

```
(python3.5)$ cat docstring_err.py
```

```
def addn(first, second, *args):
```

```
    """Returns the sum of two or more arguments:
```

```
        >>> addn(*range(10))
```

```
        55
```

```
    """
```

```
    s = first + second
```

```
    for item in args:
```

```
        s += item
```

```
    return s
```

```
(python3.5)$ python -m doctest docstring_err.py
```

```
*****
```

```
File "./docstring_err.py", line 3, in docstring_err.addn
```

```
Failed example:
```

```
    addn(*range(10))
```

```
Expected:
```

```
    55
```

```
Got:
```

```
    45
```

Hands-on

Put meaningful docstrings in the divisor function(s).

Functional programming

Algorithms based on functions,
sequences and iteration

- Everything is an object!
- Functions as function arguments
- Lambda functions
- Functional programming
- List comprehension
- Set comprehension
- Dict comprehension

First-class citizens

In Python all objects are **first-class citizens**. This means that every object supports all the language operations, such as:

- be passed to functions as arguments
- be returned by a function
- be referred by a symbolic name
- be contained in a container
- ...

In Python **everything is an object** (functions and types too).

Functions as arguments

The argument of a function can be another function.

As an example, the `list.sort()` method allows to specify the sorting algorithm. By default strings are sorted according to the lexicographic order:

```
>>> lst = ["abc", "d", "efgh", "ij"]
```

```
>>> lst.sort()
```

```
>>> lst
```

```
['abc', 'd', 'efgh', 'ij']
```


The `list.sort()` function

An alternative ordering can be applied by defining a function which returns the value to be used to sort every sequence items. For instance, if we want to sort with respect to the string length:

```
>>> def item_length(x):  
...     return len(x)  
  
...  
  
>>> lst.sort(key=item_length)  
  
>>> lst  
['d', 'ij', 'abc', 'efgh']
```

The `list.sort()` function

There is no need to define a new function: the builtin `len()` function can be directly used:

```
>>> lst.sort(key=len)
```

```
>>> lst
```

```
['d', 'ij', 'abc', 'efgh']
```

```
>>>
```

The `list.sort()` function

Often the key function is very simple but not directly available; for instance, consider

```
>>> lst = [(1, 3), (2, 5), (1, 5), (2, 6), (-2, 3), (-3, 9)]
```

We want to sort this list with respect to the sum of the two values of each item, which is a 2-tuple.

```
>>> def itemsum(x):
```

```
...     return x[0] + x[1]
```

```
>>> lst.sort(key=itemsum)
```

```
>>> lst
```

```
[(-2, 3), (1, 3), (1, 5), (-3, 9), (2, 5), (2, 6)]
```

Lambda functions

In such cases writing a new function and naming it is too much; notice that the function is not really generic, and will probably it will be used only once.

Python offers an alternative to define simple, single-line functions consisting of an expression.

```
>>> lst.sort(key=lambda x: x[0] + x[1])
```

```
>>> lst
```

```
[(-2, 3), (1, 3), (1, 5), (-3, 9), (2, 5), (2, 6)]
```

```
>>>
```

Lambda functions

A lambda function is an unnamed single-line function consisting of an expression whose value is directly returned. It cannot contain assignments or control flow statements.

A lambda function can have arguments with default values, arbitrary positional arguments, arbitrary keyword arguments, keyword-only arguments, so it has the same signature of a standard function.

```
>>> lambda x, y=10, *args, z=20, **kwargs: (x, y, z, args, kwargs)
<function <lambda> at 0x7fc3b4fa78c8>
>>>
```

Lambda functions

Lambda functions are unnamed, anyway, since everything in Python is a first-class citizen, they can be given a symbolic name:

```
>>> f = lambda x, y: x * y
```

```
>>>
```

```
>>> f(2, 5)
```

```
10
```

```
>>>
```

Functional programming is based on:

Functions

Very short functions

- simple
- a small, well defined task
- interchangeable

Sequences

The data

- containers
- generators (range)

Iteration

The glue

- uniform access to data structures

Two main functions

filter

Filter sequence items

- Select items according to a (simple) function

map

Apply a function to every item

- Create a new sequence by applying a function to the elements of the original sequence

Filter

We want to print only even numbers in a tuple:

```
>>> tpl = (2, 3, 5, 7, 8, 11, 13)
```

```
>>> for item in filter(lambda x: x % 2 == 0, tpl):
```

```
...     print(item)
```

```
...
```

```
2
```

```
8
```

```
>>>
```

Map

We want to print the cube of the numbers in a tuple:

```
>>> tpl = (2, 11, 13)
```

```
>>> for item in map(lambda x: x ** 3, tpl):
```

```
...     print(item)
```

```
...
```

```
8
```

```
1331
```

```
2197
```

Other functions

Other functions are available:

- `functools.reduce(...)`: reduce a sequence to a scalar
- `sum(sequence, start=0)`: return the sum of the sequence
- `min(sequence)`: return the minimum item of the sequence
- `max(sequence)`: return the maximum item of the sequence
- `all(sequence)`: return True if all the items of sequence evaluate to True
- `any(sequence)`: return True if any of the items of sequence evaluates to True

Hands-on

Given a list of strings, print the length of all the strings containing the letter 'a'.

Functions working on sequences

Many builtin functions work on sequences.

The following two functions are

- `enumerate(sequence)`
- `zip(*sequences)`
- The `itertools` module

The enumerate function

Sometimes, when iterating on a sequence, we need both the item and its index. In this case we can use `enumerate`, which yields a 2-tuple (`index`, `item`) for each `item` in the sequence:

```
>>> def find(value, sequence):  
...     for index, item in enumerate(sequence):  
...         if item == value:  
...             return index
```

The zip function

Sometimes, when need to iterate on two or more sequences at a time. In this case we can use `zip`, which takes n sequences as input, and yields an n -tuple ($\text{item}_0, \text{item}_1, \dots, \text{item}_n$):

```
>>> def first_equal(sequence0, sequence1):  
...     for item0, item1 in zip(sequence0, sequence1):  
...         if item0 == item1:  
...             return item0
```

Iteration stops when iteration on one of the input sequences stops.

List comprehension

List comprehension is a compact way to create lists:

```
>>> [i ** 2 for i in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> [i for i in range(10) if i % 2 == 0]
```

```
[0, 2, 4, 6, 8]
```

```
>>> [i ** 2 for i in range(10) if i % 2 == 0]
```

```
[0, 4, 16, 36, 64]
```


List comprehension

```
>>> [(i, j) for i in range(3) for j in range(4)]
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2,  
0), (2, 1), (2, 2), (2, 3)]
```

```
>>>
```

Hands-on

Given a list of strings, print the length of all the strings containing the letter 'a' using list comprehension.

Set comprehension

Set comprehension is very similar to list comprehension:

```
>>> [i - i % 5 for i in range(10)]
```

```
[0, 0, 0, 0, 0, 5, 5, 5, 5, 5]
```

```
>>> {i - i % 5 for i in range(10)}
```

```
{0, 5}
```

```
>>>
```

Dict comprehension

Dict comprehension is also available:

```
>>> {i: i % 3 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 2, 3: 0, 4: 1}
```

```
>>>
```

String formatting

Creating nice strings from
template strings and values

- Old-style string formatting
- New-style string formatting
- String templates

Three formatting alternatives

- Old-style: C-like formatting with %
- New-style: a *pythonic* alternative
- String templates: shell-like formatting with `$variable` expansion

An incomplete guide to old-style string formatting

Old-style string formatting has a substitution syntax similar to the `printf()` C function:

```
>>> "The answer to the ultimate question is %d" % 42
```

```
'The answer to the ultimate question is 42'
```

```
>>>
```

Format specification

Format specifications are available for all builtin types; for instance, %f is used to substitute floating point numbers.

The format specifier accepts width and precision (for floating point numbers); for instance, %10.2f specifies that

- A floating point number has to be substituted (f)
- The substituted string must be at least 10 characters long (10)
- Exactly 2 decimal values must be shown (10)

```
>>> "%10.2f" % 5.3
```

```
'      5.30'
```


Format specification

The %s format specifier is substituted by the object converted using `str()`.

```
>>> "%s" % "alpha"
```

```
'alpha'
```

```
>>>
```

The %r format specifier is substituted by the object converted using `repr()`.

```
>>> "%r" % "alpha"
```

```
"'alpha'"
```

```
>>>
```

Multiple positional substitutions

A template string can have multiple % substitution fields; in this case, after the % operator a tuple is expected:

```
>>> "%d %f" % (2, 3.1)
```

```
'2 3.100000'
```

```
>>>
```

Multiple keyword substitutions

Substitution fields can be passed by name; in this case a dict is expected:

```
>>> "%(num)d %(val)f" % {'val': 3.1, 'num': 2}
```

```
'2 3.100000'
```

```
>>>
```

New-style string formatting

New-style string formatting is based on the `str.format()` method and on `{...}` substitution fields:

```
>>> "The answer to the ultimate question is {}".format(42)
```

```
'The answer to the ultimate question is 42'
```

```
>>>
```

Format specification

Format specifications is almost the same as in old-style formatting; format specifiers are put after a colon inside the `{ . . . }` field:

```
>>> "The correct answer is {:.3f}".format(42.00179875)
```

```
'The correct answer is 42.002'
```

```
>>>
```

Multiple substitutions

Multiple substitutions can be passed positionally:

```
>>> "{:.2f} + {:.2f} = {:.2f}".format(2.3, 3.4, 5.7)
```

```
'2.30 + 3.40 = 5.70'
```

```
>>>
```

Fields can be manually selected:

```
>>> "{2:.2f} - {0:.2f} = {1:.2f}".format(2.3, 3.4, 5.7)
```

```
'5.70 - 2.30 = 3.40'
```

```
>>>
```

Substitution by name

Substitution fields can be passed by name:

```
>>> "{a:.2f} + {b:.2f} = {c:.2f}".format(a=2.3, b=3.4, c=5.7)
```

```
'2.30 + 3.40 = 5.70'
```

```
>>> dct = {'a': 2.3, 'b': 3.4, 'c': 5.7}
```

```
>>> "{a:.2f} + {b:.2f} = {c:.2f}".format(**dct)
```

```
'2.30 + 3.40 = 5.70'
```

```
>>> "{c:.2f} - {a:.2f} = {b:.2f}".format(**dct)
```

```
'5.70 - 2.30 = 3.40'
```

Conversion

The conversion field can be used to specify how to convert the object to string:

- **!s** (default): use `str()`
- **!r**: use `repr()`
- **!a**: use `ascii()` (as `repr()`, but non-ascii characters are shown as escape sequences)

```
>>> "{!s} is the BDFL".format("Guido")
```

```
'Guido is the BDFL'
```

```
>>> "The name of the BDFL is {!r}".format("Guido")
```

```
"The name of the BDFL is 'Guido'"
```

```
>>>
```


Accessing attributes and items

The format method allows attribute access:

```
>>> "re={z.real:.2f}, im={z.imag:.2f}".format(z=2.5 + 3.5j)
're=2.50, im=3.50'
>>>
```

The format method allows item access:

```
>>> "First element is {l[0]}".format(l=[5, 4, 3])
'First element is 5'
>>>
```

Template strings

Template strings use a `${...}` syntax, as UNIX environment variables

```
>>> import string

>>> t = string.Template("${name} is the BDFL")

>>> t.substitute(name="Guido")

'Guido is the BDFL'

>>>
```

Which string formatting should I use?

Always use **new-style** formatting! It's

- easier
- powerful
- pythonic

Hands-on

Given the books dictionary, for each book print a formatted line:

```
'<title>' was written by <author>
```