Baseball

None as default value

If you want to define a function that has an optional parameter, you always need to assign a default value to that parameter. However, it may happen that this default value is not known when the function is defined, and can only be determined at run-time (when the function is called). This is, for example, the case when the default value itself depends on argument that are passed to other parameters. In this case, it's a good idea to assign the value `None` as the default value when defining the function, and to assign a computed default value in the body of the function in case the value `None` was assigned to the optional parameter (meaning: no explicit value was passed to this parameter).

```
def func(first, second=None):
    if second is None:
        # assign the same value to the second parameter that was passed as an
        # argument to the first (mandatory) parameter, in case no explicit value
        # was passed to the second argument when calling the function
        second = first

    ...
```

This could not be solved in the following way

```
def func(first, second=first):

    ...
```

because at the time the function is defined, the parameter `first` has not been assigned a specific value. This is the same thing as using a variable that has not yet been defined.

You should also use `None` as a default value, if the actual default value that you want to assign has a mutable data type. It is recommended not to write

```
def func(first, second=[]):

    # NOTE: in this case a single empty list is created at the time the function
    #       is defined; because lists are mutable, the list can be modifed in
    #       place each time the function is called; usually, this is not the
    #       expected behavior

    ...
```

but to implement this in the following way

```
def func(first, second=None):

    # assign empty list as a default value
    # NOTE: now a new empty list is created each time the function is called, so
    #       this list is specific for that function call
    if second is None:
        second = []

    ...
```

If you want to assign a default value to a parameter that is fixed at the time the function is defined, and that has an immutable data type (`int`, `bool`, `string`, `tuple`, ...), you may safely assign the default value in the traditional way.

```
def func(first, second=42):

    ...
```

Rollover calendar

The `datetime` module

The datetime module from the The Python Standard Library defines a couple of new data types that can be used to represent dates (`datetime.date` objects) and periods of time (`datetime.timedelta` objects) in Python code. Here are some examples.

```
>>> from datetime import date
>>> birthday = date(1990, 10, 3)
>>> birthday = date(day=3, month=10, year=1990)
>>> birthday.day               # day is a property
3
>>> birthday.month             # month is a property
10
>>> birthday.year              # year is a property
1990
>>> birthday.weekday()         # weekday is a method !!
2
>>> today = date.today()
>>> today
datetime.date(2015, 11, 10)    # executed on October 11th, 2015
>>> from datetime import timedelta
>>> tomorrow = today + timedelta(1)
>>> tomorrow
datetime.date(2015, 11, 11)
>>> difference = tomorrow - today
>>> type(difference)
datetime.timedelta
>>> difference.days
1
```

Specific information

In the representation of `datetime.date` objects, months are indexed chronologically from 1 up to and including 12, where January is the first month (month 1) and December is the last month (month 12). In this assignment, we extend this in that we can number January also as the thirteenth month, February as the fourteenth month, and so on. The following table gives an overview of what we intend to say in this assignment by the months 12 up to and including 25.

original	new	original	new
12	12	19	7
13	1	20	8
14	2	21	9
15	3	22	10
16	4	23	11
17	5	24	12
18	6	25	1

As you can see we again have cyclic behaviour: the last month is followed by the first month. This always hints on using the module operator (`%`). However, because months are numbered starting from 1, not from 0, we cannot use the modulo operator directly because `12 % 12` equals `0`, whereas the first month (January) has index `1`.

The solution is to first convert the indexes 1 up to and including 12 into and indexing schema that runs from 0 up to and including 11, then use the modulo operator to implement cyclic behaviour on this new indexing scheme that starts from 0, and finally convert the new indexing scheme back to the original indexing scheme that started from 1. This can be done in the following way:

```
>>> month = 15
>>> (month - 1) % 12 + 1
3
>>> month = 24
>>> (month - 1) % 12 + 1
12
```

Cool serial numbers

Check data types with `isinstance`

To check whether or not a given object `o` has a given data type `t`, you can use the built-in function `type(o)` to see if it returns the data type `t` for the object `o`. However, it is better (more pythonic) to use the built-in function `isinstance(o, t)` in this case. This function returns a Boolean value that indicates whether or not the object `o` has data type `t` or a date type that is derived from the data type `t`.

```
>>> type(3) == int
True
>>> isinstance(3.14, int)
False
>>> isinstance(3.14, float)
True
>>> isinstance([1, 2, 3], list)
True
```

To check whether or not a given object `o` has one of multiple types, the following syntax can be used. All posible valid types are hereby listed in a tuple.

```
>>> isinstance(3, (str, int))
True
>>> isinstance('a', (str, int))
True
>> isinstance(['a'], (str, int))
False
```

Assigning functions to variables

In Python functions are themselves object of the data type `function`, so they can be assigned to variables just like any other object. This is handy if you have to code fragments that are exactly the same, except for the fact that at some point you need to call another function.

Say, for example, that we want to write a program that first needs to print all words containing the letter `a` from a given list of words, and then also needs to print all words containing the letter `b` from the same list of words. We could do this in the following way

```
>>> words = ['apple', 'banana', 'berry']
>>>
```

```
>>> def contains_a(word):
...     return 'a' in word
...
>>> def contains_b(word):
...     return 'b' in word
...
>>> for word in words:
...     if contains_a(word):
...         print(word)
...
apple
banana
>>> for word in words:
...     if contains_b(word):
...         print(word)
banana
berry
```

We could slightly rewrite the two `for` loops in the above code fragments

```
>>> func = contains_a
>>> for word in words:
...     if func(woord):
...         print(word)
...
apple
banana
>>> func = contains_b
>>> for word in words:
...     if func(word):
...         print(word)
banana
berry
```

so that we twice get exactly the same `for` loop. Because we want to avoid code duplication (we never want to program the exact same code twice), we can rewrite this by introducing another `for` loop that iterates over both functions

```
>>> functions = [contains_a, contains_b]
>>> for func in functions:
...     for word in words:
...         if func(word):
...             print(word)
apple           # generated during first iteration (func == contains_a)
banana
banana          # generated during second iteration (func == contains_b)
berry
```

Self-inventorying arrays

Functions that take an arbitrary number of arguments

In defining a function, you fix the number of arguments that needs to be passed when calling the function. This number corresponds to the number of parameters that is given with the definition of the function. For example, the following code snippet defines a function `sum` that takes exactly two arguments, and will return

the sum of adding the two objects that are passed to these parameters.

```
>>> def sum(term1, term2):
...     return term1 + term2
...
>>> sum(1, 2)
3
```

Say, however, that we wanted to write the function in such a way that it takes an arbitrary number of arguments and still returns the result of adding all the objects that are passed when calling the function. This can be done by preceding a parameter with an asterisk (*). This parameter will be assigned a typle containing all positional arguments that are passed to the function that are not assigned to other parameters.

```
>>> def sum(*terms):
...     total = 0
...     for term in *terms:
...         total += getal
...     return total
...
>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4)
10
```

In this case, all arguments passed to the function `sum` will be bundled in a tuple that is assigned to the local variable `terms`. It is also possible to name other parameters when defining the function, as long as the parameter carrying the asterisk closes the list of parameters. In addition, there can only be a single parameter that carries an asterisk.

For example, in the following code snippet we define a function `sum` that takes at least two arguments. The function still return the sum of all arguments passed to the function.

```
>>> def sum(term1, term2, *terms):
...     total = term1 + term2
...     for term in terms:
...         total += term
...     return totaal
...
>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
>>> sum(1, 2, 3, 4)
10
```