# Rollover calendar

## General information

### 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
```

## The billion-year war

### Specific information

To find all restriction sites, you can iterate over all possible start positions. Then, for each possible start position you can iterate over all possible stop positions. For each combination of a start and a stop position, you can then check if the intermediate sequence is palindromic and meets the length restrictions.

## Error detection

### General information

#### The `random` module

The random module from the The Python Standard Library can be used to add randomness to your Python code. Here's a selection of the functions implemented by this module.

| function | short description |
|---|---|
| `random()` | returns a random floating point number from the range $[0, 1[$ |
| `randint(a, b)` | returns a random integer from the range $[a, b]$ |
| `choice(s)` | returns a random element from the non-empty sequence `s` |
| `sample(s, k)` | returns `k` distinct elements from the sequence or set `s` |
| `shuffle(l)` | randomly shuffles the sequence $s$ in place |

Here are some examples.

```
>>> import random

>>> random.random()
0.954131645221452
>>> random.random()
0.3548429482674793
```

```
>>> random.randint(3, 10)
5
>>> random.randint(3, 10)
8

>>> aList = ['a', 'b', 'c']
>>> random.choice(aList)
'b'
>>> random.choice(aList)
'a'
>>> aList
['a', 'b', 'c']

>>> random.sample(aList, 2)
['a', 'c']
>>> random.sample(aList, 2)
['b', 'a']
>>> aList
['a', 'b', 'c']

>>> random.shuffle(aList)
>>> aList
['c', 'a', 'b']
```

### Passing mutable objects to functions

If you pass a mutable object to a function, the function may modify the object *in place*. This might be an explicit goal of the function, but sometimes it is not desirable to modify values that are passed to a function while the function is being executed.

Say, for example, that we pass a list to a function. What we actually pass to the function is a reference to that list and not a copy of the list (*call by reference* instead of *call by value*). As a result, the parameter to which the list is assigned becomes an alias for the list, and the function is able to modify the list itself (after all, lists are mutable data structures).

```
>>> def modify(aList, element):
...     aList.append(element)
...     return aList
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b', 'c']
```

In the example below, we first make a copy of the list that is passed to the function. Then we modify the copy, but not the original list. Making a copy of a list can be done for example by using *slicing* (`aList[:]`) or by using the built-in function `list` (`list(aList)`).

```
>>> def modify(aList, element):
...     copy = aList[:]
...     copy.append(element)
...     return copy
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b']
```

The Python Tutor gives a graphical representation of the difference between the above examples:

- example without copying
- example with copying

Because we no longer need the reference to the original list that was passed to the function, we may rewrite the function `modify` from the above example in the following way.

```
def modify(aList, element):
    aList = aList[:]
    aList.append(element)
    return aList
```

In this, the reference of the variable `aList` to the original list that is passed to the function `modify`, is replaced by a reference to a copy of the list. This replacement is only visible inside the function, because the variable `aList` is a local variable of the function `modify`. You can also inspect this example using the Python Tutor.