Colorful fruits

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']
```

Friday the 13th

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
```

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):

    ...
```

Route

General information

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
```

Square routes

General information

Align strings over a fixed number of positions

You can use the *format specifier* of the string method `format` to reserve a fixed number of positions to output a given text fragment. This is done by formatting the text as a string (indicated by the letter `s`), preceded by an integer that indicates the fixed number of positions that needs to be reserved for the string.

```
>>> f"{'abc':5s}"   # reserve 5 positions
'abc  '
```

In case the string is longer than the number of reserved positions, the placeholder is replaced by the entire string, and thus takes more than the reserved space. In case the string is shorter than the number of reserved positions, the string is left aligned by default. You can also explicitly define the type of alignment in the *format specifier*: left, right or centered. Python will automatically pad the string with additional spaces to the left and/or to the right.

```
>>> f"{'abc':<5s}"   # reserve 5 positions, left alignment
'abc  '
>>> f"{'abc':>5s}"   # reserve 5 positions, right alignment
'  abc'
>>> f"{'abc':^5s}"   # reserve 5 positions, centered alignment
' abc '
```

In case a centered alignment is chosen and the number of additional spaces is an odd number, Python prefers to add one more space to the right than to the left.

Initialize nested list

A rectangular grid with m rows and n columns can be represented by a list `l` containing m lists, with each list from `l` having n elements. Say, for example, that you want to create a list with three rows and two columns, with each cell of the grid containing the empty string. This can be done using *list comprehensions*.

```
>>> m = 3
>>> n = 2
>>> grid = [['' for _ in range(n)] for _ in range(m)]
>>> grid
[['', ''], ['', ''], ['', '']]
>>> grid[0][0] = 'A'
>>> grid
[['A', ''], ['', ''], ['', '']]
```

In most cases you can also use the multiplication operator `*` to create lists of a given size, where each cell is initialized with the same object. But if this object is mutable, you might get surprising results.

```
>>> m = 3
>>> n = 2
>>> [''] * m
['', '', '']
>>> grid = [[''] * n] * m
>>> grid
[['', ''], ['', ''], ['', '']]
>>> grid[0][0] = 'A'
```

```
>>> grid
[['A', ''], ['A', ''], ['A', '']]
```

As you can see, putting the string `A` in cell `grid[0][0]`, causes the string `A` to be placed as well in cells `grid[1][0]` and `grid[2][0]`. The reason for this strange behaviour is that the operator `*` has created aliases to the same list, such that `grid[0]`, `grid[1]` and `grid[2]` all reference exactly the same list object. As a result, adjusting one of these lists, changes all of them (well, actually, there is only one of them). This can be seen clearly when you use the Python Tutor.