2.9 Inheriting properties from built-in classes

Python gives you the ability to create a class that inherits properties from any Python built-in class in order to get a new class that can enrich the parent's attributes or methods. As a result, your newly-created class has the advantage of all of the well-known functionalities inherited from its parent or even parents and you can still access those attributes and methods.

Later, you can override the methods by delivering your own modifications for the selected methods.

In the following example, we'll create an implementation of our own list class, which will only accept elements of the integer type. But, wait – why might you need such an object?

Imagine that you need to collect the serial numbers of sold tickets. Sound reasonable enough?

Your new class will be based on the Python list implementation and will also validate the type of elements that are about to be placed onto it.

Such a list can be used in an application that requires the list elements to be of a specific type (integers in the ticketing example), and control over the types of elements is given to the mechanisms of the new class.

As a result, when solving a domain problem, we focus on the problem and not on type control.

Look at the code presented in the editor pane.

```
class IntegerList(list):

    @staticmethod
    def check_value_type(value):
        if type(value) is not int:
            raise ValueError('Not an integer type')

    def __setitem__(self, index, value):
        IntegerList.check_value_type(value)
        list.__setitem__(self, index, value)

    def append(self, value):
        IntegerList.check_value_type(value)
        list.append(self, value)

    def extend(self, iterable):
        for element in iterable:
            IntegerList.check_value_type(element)

        list.extend(self, iterable)


int_list = IntegerList()

int_list.append(66)
int_list.append(22)
print('Appending int elements succeed:', int_list)

int_list[0] = 49
print('Inserting int element succeed:', int_list)
```

```
int_list.extend([2, 3])
print('Extending with int elements succeed:', int_list)

try:
    int_list.append('8-10')
except ValueError:
    print('Appending string failed')

try:
    int_list[0] = '10/11'
except ValueError:
    print('Inserting string failed')

try:
    int_list.extend([997, '10/11'])
except ValueError:
    print('Extending with ineligible element failed')

print('Final result:', int_list)
```

Something that's worth commenting on is that we have delivered:

- a static, dedicated method for checking argument types. As we have delegated this responsibility to only one method, the code will be shorter, cleaner and easier to maintain. We'll make use of this method a few times. In case the argument's type is not an integer, a ValueError exception is raised;
- an overridden method `__setitem__`, which is a magic method (mind the underscores) responsible for inserting (overwriting) an element at a given position. This method calls the check_value_type() method and later calls the genuine method `__setitem__` which comes from the parent class, which does the rest of the job (sets the validated value at a given position). Now you can sigh – "oh, what a great ability!"
- an overridden method, append(), which is responsible for appending an element to the end of the list. This method follows the previous way of dealing with a new element;
- an overridden method, extend(), to verify and add a collection of elements to the object.

What have we not delivered?

- All the remaining methods have remained unchanged, so our new list-like class will still behave like its parent in those places.

To make our newly-created class fully functional, it's necessary to deliver implementations for the methods:

- insert(index, object)
- `__add__`()

These implementations should be fairly similar to the implementations delivered above (validate the type and then call the corresponding superclass method).

output

```
Appending int elements succeed: [66, 22]
Inserting int element succeed: [49, 22]
Extending with int elements succeed: [49, 22, 2, 3]
Appending string failed
Inserting string failed
Extending with ineligible element failed
Final result: [49, 22, 2, 3]
```

In the next example, we'll create a class based on Python's built-in dictionary, which will be equipped with logging mechanisms for details of writing and reading operations performed on the elements of our dictionary.

```
from datetime import datetime


class MonitoredDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.log = list()
        self.log_timestamp('MonitoredDict created')

    def __getitem__(self, key):
        val = super().__getitem__(key)
        self.log_timestamp('value for key [{}] retrieved'.format(key))
        return val

    def __setitem__(self, key, val):
        super().__setitem__(key, val)
        self.log_timestamp('value for key [{}] set'.format(key))

    def log_timestamp(self, message):
        timestampStr = datetime.now().strftime("%Y-%m-%d (%H:%M:%S.%f)")
        self.log.append('{} {}'.format(timestampStr, message))


kk = MonitoredDict()
kk[10] = 15
kk[20] = 5

print('Element kk[10]:', kk[10])
print('Whole dictionary:', kk)
print('Our log book:\n')
print('\n'.join(kk.log))
```

In other words, we are arming a Python dictionary with the ability to log details (time and operation type) of:

- class instantiation;
- read access;
- new element creation or update.

A few notes for the code implementing the MonitoredDict class:

- we have subclassed a dict class with a new init() method that calls the init() method from its super class. Additionally, it creates a list (self.log) that plays the role of a log book. Finally, the log book is populated with a message noting that the object has been created;
- we have created the log_timestamp() method that appends crucial information to the self.log attribute;
- we have overridden two methods inherent for the dictionary class (getitem() and setitem()) to deliver a richer implementation that logs activities. But don't worry, we're not losing anything from the parent dictionary class, because we're still calling the corresponding methods.

As you run the code, you'll see that the new class is compatible with its parent, so you can use it in your applications that require activity tracking.

output

```
Element kk[10]: 15
Whole dictionary: {10: 15, 20: 5}
Our log book:

2023-10-23 (13:05:29.321025) MonitoredDict created
2023-10-23 (13:05:29.321062) value for key [10] set
2023-10-23 (13:05:29.321078) value for key [20] set
2023-10-23 (13:05:29.321092) value for key [10] retrieved
```

How about implementing such a “history recording” feature in a banking application?

In the following slide, we'll examine another feature that could be useful in a banking app.

The IBAN Only Dictionary

The “Integer only list” is an example of the employment of a subclassed built-in list to check the types of elements being added to the list. How about checking the values of the keys being used when new elements are added to the dictionary?

For those of you who have taken the course “Programming Essentials in Python”, the IBAN Validator should be well know. But, if you haven’t taken the course, now is a good moment familiarize yourself with IBAN and ways to validate it.

We'll use the IBAN Validator to ensure that our banking app dictionary contains only validated IBANs (keys) and info about the associated balance (value).

Well, what is this IBAN then?

IBAN is an algorithm used by European banks to specify account numbers. The standard name **IBAN** (International Bank Account Number) provides a simple and fairly reliable method of validating the account numbers against simple typos that can occur during rewriting of the number, e.g., from paper documents, like invoices or bills, into computers.

You can find more details here: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

An IBAN-compliant account number consists of:

- a two-letter country code taken from the ISO 3166-1 standard (e.g., FR for France, GB for the United Kingdom, DE for Germany, and so on)
- two check digits used to perform the validity checks – fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good;
- the actual account number (up to 30 alphanumeric characters – the length of that part depends on the country)

The standard says that validation requires the following steps (according to Wikipedia):

- (step 1) Check that the total IBAN length is correct as per the country (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
- (step 2) Move the four initial characters to the end of the string (i.e., the country code and the check digits)
- (step 3) Replace each letter in the string with two digits, thereby expanding the string, where A = 10, B = 11 ... Z = 35;
- (step 4) Interpret the string as a decimal integer and compute the remainder of that number on division

by 97; If the remainder is 1, the check digit test is passed and the IBAN might be valid.

Look at the genuine code in the editor. In a moment we'll modify it, but first let's analyze it:

- line 03: ask the user to enter the IBAN (the number can contain spaces, as they significantly improve number readability...
- line 04: ...but remove them immediately;
- line 05: the entered IBAN must consist of digits and letters only – if it doesn't...
- line 06: ...output the message;
- line 07: the IBAN mustn't be shorter than 15 characters (this is the shortest variant, used in Norway)
- line 08: if it is shorter, the user is informed;
- line 09: moreover, the IBAN cannot be longer than 31 characters (this is the longest variant, used in Malta)
- line 10: if it is longer, make an announcement;
- line 11: start the actual processing;
- line 12: move the four initial characters to the number's end, and convert all letters to upper case (step 02 of the algorithm)
- line 13: this is the variable used to complete the number, created by replacing the letters with digits (according to the algorithm's step 03)
- line 14: iterate through the IBAN;
- line 15: if the character is a digit...
- line 16: just copy it;
- line 17: otherwise...
- line 18: ...convert it into two digits (note the way it's done here)
- line 19: the converted form of the IBAN is ready – make an integer out of it;
- line 20: is the remainder of the division of iban2 by 97 equal to 1?
- line 21: if yes, then success;
- line 22: otherwise...
- line 23: ...the number is invalid.

Let's add some test data (all these numbers are valid – you can invalidate them by changing any character).

- British: **GB72 HBZU 7006 7212 1253 00**
- French: **FR76 30003 03620 00020216907 50**
- German: **DE02100100100152517108**

Now, let's add a new exception class and wrap the previous IBAN validating snippet into a function, reformulate the last condition, and use it as a helper function.

```
class IBANValidationError(Exception):
    pass


def validateIBAN(iban):
    iban = iban.replace(' ', '')

    if not iban.isalnum():
        raise IBANValidationError("You have entered invalid characters.")

    elif len(iban) < 15:
        raise IBANValidationError("IBAN entered is too short.")

    elif len(iban) > 31:
        raise IBANValidationError("IBAN entered is too long.")

    else:
        iban = (iban[4:] + iban[0:4]).upper()
```

```
        iban2 = ''
        for ch in iban:
            if ch.isdigit():
                iban2 += ch
            else:
                iban2 += str(10 + ord(ch) - ord('A'))
        ibann = int(iban2)

        if ibann % 97 != 1:
            raise IBANValidationError("IBAN entered is invalid.")

        return True


test_keys = ['GB72 HBZU 7006 7212 1253 01', 'FR76 30003 03620 00020216907 50',
'DE02100100100152517108' ]

for key in test_keys:
    try:
        print('Status of "{}" validation: '.format(key))
        validateIBAN(key)
    except IBANValidationError as e:
        print("\t{}".format(e))
    else:
        print("\tcorrect")
```

To sum up, our validateIBAN(iban) function:

- requires a parameter; it is a string to check whether it contains an IBAN-compliant account number;
- raises an IBANValidationError exception when the supplied string carries an incorrectly formulated account number;
- returns a True value when the account number conforms to all IBAN requirements.

Finally, let's add a loop to check three example IBANs. Pay attention to the fact that the first IBAN value has been modified to raise an exception.

output

```
Status of "GB72 HBZU 7006 7212 1253 01" validation:
    IBAN entered is invalid.
Status of "FR76 30003 03620 00020216907 50" validation:
    correct
Status of "DE02100100100152517108" validation:
    correct
```

Having a validateIBAN() function in place, we can write our own class that inherits after a built-in dict class.

```
class IBANDict(dict):
    def __setitem__(self, _key, _val):
        if validateIBAN(_key):
            super().__setitem__(_key, _val)

    def update(self, *args, **kwargs):
        for _key, _val in dict(*args, **kwargs).items():
            self.__setitem__(_key, _val)
```

In this implementation, we have delivered the method setitem() which calls validateIBAN() and on success calls the genuine setitem() method. This piece of code is responsible for statements like:

```
my_dict[key] = value
```

We have also delivered the method update() which iterates the parameters passed, and for each correct pair calls the __setitem__() method

After joining all the snippets, we get the following code:

```
import random


class IBANValidationError(Exception):
    pass


class IBANDict(dict):
    def __setitem__(self, _key, _val):
        if validateIBAN(_key):
            super().__setitem__(_key, _val)

    def update(self, *args, **kwargs):
        for _key, _val in dict(*args, **kwargs).items():
            self.__setitem__(_key, _val)


def validateIBAN(iban):
    iban = iban.replace(' ', '')

    if not iban.isalnum():
        raise IBANValidationError("You have entered invalid characters.")

    elif len(iban) < 15:
        raise IBANValidationError("IBAN entered is too short.")

    elif len(iban) > 31:
        raise IBANValidationError("IBAN entered is too long.")

    else:
        iban = (iban[4:] + iban[0:4]).upper()
        iban2 = ''
        for ch in iban:
            if ch.isdigit():
                iban2 += ch
            else:
                iban2 += str(10 + ord(ch) - ord('A'))
        ibann = int(iban2)

        if ibann % 97 != 1:
            raise IBANValidationError("IBAN entered is invalid.")

        return True


my_dict = IBANDict()
```

```
keys = ['GB72 HBZU 7006 7212 1253 00', 'FR76 30003 03620 00020216907 50',
'DE02100100100152517108']

for key in keys:
    my_dict[key] = random.randint(0, 1000)

print('The my_dict dictionary contains:')
for key, value in my_dict.items():
    print("\t{} -> {}".format(key, value))

try:
    my_dict.update({'dummy_account': 100})
except IBANValidationError:
    print('IBANDict has protected your dictionary against incorrect data
insertion')
```

This is a good example of making use of the subclassing feature to enrich your code with an important set of syntax and semantic checks.

To test the efficiency, add and try running the following code:

```
try:
    my_dict.update({'dummy_account': 100})
except IBANValidationError:
    print('IBANDict has protected your dictionary against incorrect data
insertion')
```

The output should make your eyes happy.

output

```
The my_dict dictionary contains:
    GB72 HBZU 7006 7212 1253 00 -> 683
    FR76 30003 03620 00020216907 50 -> 240
    DE02100100100152517108 -> 114
IBANDict has protected your dictionary against incorrect data insertion
```

Sumary

- Python allows you to subclass any built-in class such as a list, tuple, dictionary, and many others;
- by subclassing the built-ins, you can easily adapt generics to provide more sophisticated features;
- by subclassing the built-ins, you can modify only the parts (methods, attributes) that you intend to modify, while all remaining parts will behave as good old built-ins.