Sequence Template Class

Yiqiao Yin
W.Y.N. Associates, LLC

Abstract

The assignment creates a report on this useful class: template class. The report introduces the template class (1 page). Then it discusses the usage and the real world application with an example (1 page). Next, the report covers the benefits and drawbacks of using template classes (1 page). In responding to some drawbacks of, the report also discusses some alternatives to template classes (1 page). The report also presents a code example of how this class is used in Python. In addition, existing research that can be conducted using this template classes is discussed (1 page) along with future research opportunities (1 page). The language of choice is python for this assignment. Though sequence class exists in many programming languages, the sequence types and the syntax are concentrated in python programming language.

1 What are template classes?

One of the most popular computer languages today is Python Van Rossum and Drake Jr (1995); Van Rossum (1995). Python is a general-purpose computer programming language and the design shows significant readability with the corresponding indentation. The construction of the language is object-oriented. It is designed to produce clear, logical expression for small and large industrial projects. This section introduces the template class in strings and also introduces the fundamental building blocks of three major sequences in python.

1.1 String Template

The most basic template to introduce is the string template.Morin (2013). The string template is a library under the "string" package. The library allows the construction of any arbitrary string syntax where the input parameter or variable of the string syntax are identified by the placeholder of a dollar sign, i.e. "$". There are two unique design methods in string template. The first is to use curly brackets, i.e. { }. The usage of curly brackets allow users to change what is inside of the curly brackets while adding template right after it. Another interesting method is to follow the placeholder with another dollar sign, i.e. $$. This allows the interpreter to output the actual character of the dollar sign.

```
# import the library
from string import Template

# create a basic template structure
# the syntax after the "$" symbol is allowed for change
```

```
template = Template('$name wrote a report that has the title:
    ↪ $title_name.')

# change the syntax using .substitute method
template.substitute(name="Yiqiao", title_name="Sequence Template Class
    ↪ ")
"Yiqiao wrote a report that has the title: Sequence Template Class."

# store the information of the new output in an object
new_output = template.substitute(name="Yiqiao", title_name="Sequence
    ↪ Template Class")

# display the output
new_output
"Yiqiao wrote a report that has the title: Sequence Template Class."
```

A more thoughtful design is to inject the numerical input into a string template. In this design, it is important to notice that a numerical input such as the number “12” in the example below needs to be processed as a string. We use a wrapper function of “str()” to ensure that the data type of “12” is string. This is because we want to enter the number “12” into the template that is a string. Another small trick here is to use the double dollar sign $$ to escape the actual dollar symbol.

```
# create another template using double dollar sign to escape the actual
    ↪  dollar sign symbol
# and it is followed with a third dollar sign because it is the
    ↪ placeholder for the
# variable "money_amount"
template = Template('$name bought Chinese food for lunch using
    ↪ $$$money_amount dollars.')

# enter desired object names for the parameter
# notice that if the numerical input 12 is float or integer
# it will not be stored unless it is passed in as a string
template.substitute(name="Yiqiao", money_amount=str(12))

"Yiqiao bought Chinese food for lunch using $12 dollars."
```

A more complicated design is to implement the template in a for loop to allow larger-scale of data propagate. In this case, we can provide the program a data structure that is a list of tuples of two inputs. The two inputs are a string and a number (in python it is called float). Then we can create a string template that has input parameters that are stored in the list object. In this case, to populate all the items in the list object into the template, we use a for loop to achieve this goal. We can print out the updated object of the template in the for loop. The example is done below.

```
# design a list to store information
class_grades = [('course 8101', 100), ('course 8110', 90)]
print(class_grades)

# design template and store in an object called class_grades_template
class_grades_template = Template('This student takes the $class_name
    ↪ and receives grade of $grade.')

# use for loop to print the information
for i in class_grades:
    print(class_grades_template.substitute(class_name = i[0], grade=i
        ↪ [1]))

This student takes the course 8101 and receives grade of 100.
This student takes the course 8110 and receives grade of 90.
```

2 Introduction of Sequences

There are three major types of sequence objects in python. They are list, tuple, and range. These are the basic building block of the python programming language. In addition, many other sequence types of built and tailored to particular problems and algorithms, yet most of the modern day development of sequence objects are built upon the idea of list, tuple, and range. This subsection we introduce all of them.

2.1 Lists

The first sequence type is called list. The list is carried out by the syntax “[]” in python interpreter. The values or items are sequentially laid out in the syntax “[]”. Values and strings can be stored in a list and the items in a list has sequential order in it. The following we present some sample code to get started with list objects. Empty list with [] can exist by itself. In addition, we can also store strings (such as letters ’a’, ’b’) and numbers (such as the digit 1) in the list. We can print out the list using a for loop so the entries can be displayed one by one. We can also use a for loop tracking index 0, 1, 2 and so on to extract the information from the list. The items are stored and they have certain orders in a list starting from the index 0. In other words, the index 0 would be the first entry of a list.

```
empty_list = []
empty_list
[] # output

a = [’a’, ’b’, 1]
print(a)
[’a’, ’b’, 1] # output

# for loop
for i in a:
    print(i)
a
b
1

# for loop
for i in [0,1,2]:
    print(a[i])
a
b
1
```

There can also be nested lists. This means that we can design a list with another list inside. The following shows a few examples of how the nested list works. In the following case, a list “b” is defined with some arbitrary inputs that are also lists. For example, the first sub-list is a string of three letters (“a”, “b”, and “c”) and the second sub-list is a list of float (1 and 2). The list “b” can be displayed. We can append a new list inside of the list b by using the “.append” method.

This is convenient when we want to populate the items inside the list. The “.append” method will add the new item into the list by placing it in the end. In other words, originally the list “b” has two entries with the index 0 and 1. Now the updated list “b” has a third entry.

```
b = [['a','b','c'],[1,2]]
print(b)
[['a', 'b', 'c'], [1, 2]]

b.append(['ab',12])
print(b)
[['a', 'b', 'c'], [1, 2], ['ab', 12]]
```

The data type list is mutable. A list (such as the examples show above) has the following important characteristics:

1. Elements can be modified;
2. Individual values can be replaced;
3. The order of elements can be changed. [1]

These characteristics make the list a very dynamic and powerful tool to have in a coding project. More examples can be found to manipulate a list. A delete action can be executed on a list. We can erase the first entry by using “del” function. The items in a list can also be traced using index. For example, a particular subset of the list can be extracted using index 0, 1, 2, and so on. Consecutive entries of a list can also be extracted using “:” symbol to indicate a range under a list. For example, the first two and the second entry can be extracted using “list_name[0:1]” syntax for a list. Another interesting property comes with mutability is the fact that we can display entries backwards or reverse a list easily. A display can be done in reverse order of the entries in the list by using “-” sign. For example, “-1” means the last entry in the list while “-2” means the second to the last entry in the list. A range counting from the last entry of the list can be done using “-0:2” while the digit on the left of the “:” sign is inclusive and the digit on the right of the “:” sign is exclusive.

```
# recall we have defined a list "b"
print(b)
[['a', 'b', 'c'], [1, 2], ['ab', 12]]

# delete the first entry
del b[0]
print(b)
[[1, 2], ['ab', 12]]

print(b[0])
[1, 2]
print(b[0:1])
[[1, 2]]
print(b[0:2])
[[1, 2], ['ab', 12]]
print(b[0:3])
[[1, 2], ['ab', 12]]
print(b[0::])
[[1, 2], ['ab', 12]]

```

[1] Source: Lists: Mutable and Dynamic.

```
print(b[-1])
[’ab’, 12]
print(b[-2])
[1, 2]
print(b[-0:2])
[[1, 2], [’ab’, 12]]
print(b[::-1])
[[’ab’, 12], [1, 2]]
```

2.2 Tuples

The tuples are another famous type of sequence. Tuples are designed to store collections of heterogeneous data[2]. Tuples work like a list in basic ways. Tuples can be constructed using “()” syntax. Tuples can also be sliced and the entries can be extracted using index just like the Lists. However, if we are trying to break up a tuple and reassign an entry in the tuple to another input assignment, this cannot be done and we will receive an error message. An example is shown below and the attempt is to try to reassign the second entry of the tuple named “c” to another integer 3. In doing so, we received an error message telling us the object does not support such assignment.

```
c = (1,2)
print(c)
(1, 2)

c[1] = 3
TypeError: ’tuple’ object does not support item assignment
```

As a remark, it is worth noting that the immutable data values are a data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error. A mutable data value is a data value that can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.[3]

From an article on Towards Data Science[4], a key difference is the size of a tuple and a list with the exact same entry. This difference is shown from the memory created on a list and a tuple with the exact same entry ’yiqiao’ and ’yin’ (which is shown in the code below). The memory created using list is 88 while the memory created using tuple is 72. The search time is also different between a tuple and a list. For example, we can create a list and a tuple with a million integers inside both of them. We can create a loop search for each entry of the list and the tuple to check if they equal to the digit 1. The time consumption can be recorded using “time” package in python. The print statements are displayed in the end of the example using the “time” package. There is a 0.016 second difference between for loops using list and tuple.

```
import sys

# create empty placeholder
a_list = list()
a_tuple = tuple()

# define objects
a_list = [’yiqiao’, ’yin’]
```

[2] Source: Python Document

[3] Source: OpenBookProject Net.

[4] Medium article: source.

```
a_tuple = ('yiqiao', 'yin')

# print size
print(sys.getsizeof(a_list))
print(sys.getsizeof(a_tuple))
88
72

import time

a_list = list(range(0, 1000000))
a_tuple = tuple(range(0, 1000000))
print(len(a_list), len(a_tuple))
1000000 1000000

start_time = time.time()
for i in a_list:
    1 == i
end_time = time.time()
print("Time consumed for LIST: ", end_time - start_time)
Time consumed for LIST: 0.08094334602355957

start_time = time.time()
for j in a_tuple:
    1 == j
end_time = time.time()
print("Time consumed for TUPLE: ", end_time - start_time)
Time consumed for TUPLE: 0.09636068344116211
```

A remark that is worth noting is that though the lists data type is more flexible than the tuples data type an occasion can arise where the data should not be changed (such as a hash table). This is a situation when tuples can be preferred.

3 Real-world Example

In this section, we create a real world example to demonstrate some key components of lists, tuples, and template that we introduced in the first section. We create a small project called the "Course Schedule" project where a program is designed to allow me to add or subtract a course as a PhD student.

First, the example starts with a motivation example. The target is to be able to create a function to add and delete courses in a student's schedule. A course can have multiple information. For example, records can show the title, course ID, department, credit hours, and level of difficulty of a course. A course is coded as a list. The information can be stored in a list and we can append this list to a bigger list called "list_of_courses". The big list "list_of_courses" is the overall list of courses that we can add or reduce classes. To build up the intuition, the following code is written.

```
# imagine we are creating a time table for my schedule at NCU
# the unit of analysis here is a computer science course
list_of_courses = []

# add a course
# a course can have a title, department where it is hosted, course
    ↪ credits, level of courses
```

```
list_of_courses.append(['algorithm', 12345, 'computer science', 4, '
    ↪ undergrad'])
print(list_of_courses)
[['algorithm', 12345, 'computer science', 4, 'undergrad']]

# add a new course
list_of_courses.append(['optimization', 23456, 'mathematics', 5, 'grad
    ↪ '])
print(list_of_courses)
[['algorithm', 12345, 'computer science', 4, 'undergrad'], ['
    ↪ optimization', 23456, 'mathematics', 5, 'grad']]

# add a new course
list_of_courses.append(['machine learning', 14253, 'computer science',
    ↪ 4, 'master'])
print(list_of_courses)
[['algorithm', 12345, 'computer science', 4, 'undergrad'], ['
    ↪ optimization', 23456, 'mathematics', 5, 'grad'], ['machine
    ↪ learning', 14253, 'computer science', 4, 'master']]

# reduce an existing course
del list_of_courses[0]
print(list_of_courses)
[['optimization', 23456, 'mathematics', 5, 'grad'], ['machine learning',
    ↪  14253, 'computer science', 4, 'master']]
```

We can also create a template to print out or display the information we have stored in the data. The template makes the process very efficient because a for loop will allow us to print out the course information in natural language and.

```
template = Template('The $course_title has $course_ID and it is under
    ↪ the $dept_name department with $num_credit credit hours at a
    ↪ $level level.')

for i in list_of_courses:
    print(template.substitute(course_title=i[0], course_ID=i[1],
        ↪ dept_name=i[2], num_credit=i[3], level=i[4]))
The optimization has 23456 and it is under the mathematics department
    ↪ with 5 credit hours at a grad level.
The machine learning has 14253 and it is under the computer science
    ↪ department with 4 credit hours at a master level.
```

3.1 Benefits, Drawbacks, and Alternatives

This subsection discusses the benefits and the drawbacks of the template classes.

There are many benefits with the template classes in programming. A template class is formatted and it can also be pre-formatted which is very efficient when it comes to streamlining the production code. It can also work with many different types of sequences such as lists, tuples, and ranges. In addition, template class can be easily implemented in definition or class objects which allows us to further utilize its tools and useful functionalities. The input of a template can also be implemented using a hash symbol, i.e. #, which allows different language environment. Another benefit discussed above is that usage of the double dollar sign symbol which allows the escape of the special character. One major reason to use template is the capability to write extremely efficient and neat-looking code. Such ability can be extended drastically using for loops or while loops. In our example introduced in the previous section, we implemented many examples using for loop and the program is able to display natural language with desired input without having to type out the inputs more than one time. Such capabilities provide great benefits to use template to start the coding project.

Despite many of the successes we discussed above, there are some drawbacks of template class. Like many other functions in major programming languages, one drawback is the key error. This is due to the nature of the design of the template class, which requires a key. Without a key identification or provided with an incorrect key identification, a template class cannot execute. Since the template class usually works with strings, numerical value can sometimes cause value error. In value error, invalid character can cause placeholder to misrepresent what is desired to output. This results in a bad placeholder. Another major drawback is python compiler actually allows very handy alternatives to the template class. The following example is presented to show the python alternative when in the absence of the template class. As shown below, the curly bracket, {}, can be introduced when a user desires to achieve the same goal but in the absence of the template class. In fact, more than one approach can be carried out using the curly bracket. Below we show two separate examples of achieving the same goal without the template class.

```
t = Template(’$name is happy today doing homework $num from NCU.’)
print(t.substitute(name=’Yiqiao’, num=3))
Yiqiao is happy today doing homework 3 from NCU.

t = "{} is happy today doing homework {} from NCU."
print(t.format("Yiqiao", 3))
Yiqiao is happy today doing homework 3 from NCU.

name = "Yiqiao"
num = 3
print(f"{name} is happy today doing homework {num} from NCU.")
Yiqiao is happy today doing homework 3 from NCU.
```

4 Existing Research

This section reviews some of the existing literature that is related or built upon the idea of template classes.

The template class discussed in this report is a generic programming procedure in languages such as C++ or Python. In most situation discussed in this report, template class has been used to create efficient programming pipeline such as providing benchmark or streamlining the production code using certain template. However, the concept of template class can be extracted and use in many different places. Specifically, in this report, we focus on sequence types such as lists and tuples. In practice, lists, in particular, can also be extended into tensors which can take high-dimensional shapes. Instead of building production code using template classes, a template benchmark can be created in a 2D array to search for important patterns in another 2D array. Techniques such as these are extremely important and helpful at extracting useful information from image sequences. Template matching is one of the most common techniques in computer vision and signal and image processing. A general framework for object tracking in video images is proposed by Jurie and Dhome (2001) of which the methodology consists in low-order parametric models for the image motion of a target. Their proposed algorithm allows to track in real time which is a big step from the traditional template matching. Fast-Match is another efficient algorithm for approximate template matching under 2D affine transformations that minimizes the Sum-of-Absolute-Differences error measure, proposed by Korman et al. (2013). Algebraic template matching proposed by Omachi and Omachi (2007) calculates the similarities between the template and the partial images of the input image, for various widths and heights. In their algorithm, a polynomial that approximates the template image is used to match the input image instead of the template image.

5 Future Research

The future of template matching depends on the field of application. From the algorithmic perspective, it is clear that the field has been consolidated. It would be very challenging to come up with a new template class to does the job more efficiently using a famous language such as python. However, the concept of template classes deserve much more attention. The field of signal and image processing is a great example to illustrate the work of template. In this case, images are common assumed to be in the shapes of 2D array or tensor, which provided the playground for researchers to apply template ideas on a higher dimension. If there are existing research in image analysis, it is not far fetch to say that in the future research in higher dimensional data sets can also be attempted using template matching.

References

Jurie, F. and Dhome, M. (2001). A simple and efficient template matching algorithm. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 544–549. IEEE.

Korman, S., Reichman, D., Tsur, G., and Avidan, S. (2013). Fast-match: Fast affine template matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2331–2338.

Morin, P. (2013). *Open Data Structures: An Introduction*, volume 2. Athabasca University Press.

Omachi, S. and Omachi, M. (2007). Fast template matching with polynomials. *IEEE Transactions on Image Processing*, 16(8):2139–2149.

Van Rossum, G. (1995). Python reference manual. *Department of Computer Science [CS]*, (R 9525).

Van Rossum, G. and Drake Jr, F. L. (1995). *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.