Шаблонизатор Jinja2

# Что это? Использование {{ }} в шаблонах

[Смотреть материал на видео](https://www.youtube.com/watch?v=cFJqMXxVNsI&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/chto-eto-ispolzovanie-v-shablonah>

В языке Python имеется весьма полезный модуль Jinja (читается как *дзиндзя* – в честь японского синоитского святилища) служит для обработки шаблонов. Им можно пользоваться в своих собственных приложениях, но он также входит в состав множества других широко используемых модулей, например, Flask или Django. В рамках наших занятий мы рассмотрим основные возможности Jnija, которых, наверное, будет достаточно в 80%, а то и 90% случаях. Но, как всегда, полную и подробную информацию можно найти на официальном сайте:

<https://jinja.palletsprojects.com/en/2.11.x/>

Преимущество модуля Jinja в простоте использования и богатстве  предоставляемых возможностей для преобразования текстовых шаблонов. Но для начала нам потребуется его установить и, так как он входит в состав множества других пакетов, то возможно, у вас на компьютере он уже установлен. Попробуйте для начала выполнить вот такую простую программу на Python:

**from** jinja2 **import** Template

name = "Федор"

tm = Template("Привет {{ name }}")

msg = tm.render(name=name)

**print**(msg)

И если в консоли увидите строку:

Привет Федор

значит модуль Jinja готов к работе. Иначе, его следует установить с помощью команды:

pip install Jinja

Итак, давайте внимательнее посмотрим, что делает эта программа. Сначала создается объект Template (шаблон), которому передается строка. В этой строке внутри двух фигурных скобок указана переменная name. Затем идет вызов метода render из этого объекта и передается именованный параметр name. В результате метод render будет ссылаться на словарь с ключом name и значением «Федор»:

{name: Федор}

Благодаря этому, вместо name будет в шаблоне подставлено значение «Федор», а метод render возвратит преобразованную строку.

Конечно, первая реакция: зачем так городить огород, когда все это можно сделать с помощью обычных F-строк Python:

msg2 = f"Привет {name}"

**print**(msg, msg2, sep="**\n**")

Все верно, и в таких простых случаях именно F-строки или другой строковых функционал и нужно использовать. Но это был простейший пример, скорее для общего понимания принципа работы модуля Jinja. В дальнейшем мы увидим преобразования и посложнее, которые с помощью строк так просто уже не реализовать.

Итак, метод render класса Tempalte понимает следующие определения:

* {% %} – спецификатор шаблона;
* {{ }} – выражение для вставки конструкций Python в шаблон;
* {# #} – блок комментариев;
* # ## – строковый комментарий.

Только что мы видели на примере использование двух фигурных скобок. Давайте продолжим их изучение и немного усложним наш пример:

name = "Федор"

age = 28

tm = Template("Мне {{age}} лет и зовут {{ name }}.")

msg = tm.render(name=name, age=age)

**print**(msg)

Здесь все очень похоже на предыдущую программу, только используется два определения внутри шаблона с ключами age и name. Причем, значения ключей в методе render можно записывать в любом порядке. Кроме того, можно просто написать:

tm = Template("Мне {{a}} лет и зовут {{ n }}.")

msg = tm.render(n=name, a=age)

И все будет работать абсолютно также. То есть, здесь имя параметра – это ключ, вместо которого подставляется его значения при формировании выходной строки. Но, через этот ключ шаблон получает доступ к соответствующим переменным и может выполнять любые операции, известные в Python, например, так:

tm = Template("Мне {{a\*2}} лет и зовут {{ n.upper() }}.")

msg = tm.render(a=age, n=name)

На выходе получим строку:

Мне 36 лет и зовут ФЕДОР.

Как видите, внутри двойных фигурных скобок можно использовать операторы языка Python.

Продолжим усложнять наш пример и, теперь предположим, что имеется класс:

**class** Person:

**def** \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

Затем, создается экземпляр этого класса:

per = Person("Федор", 33)

И необходимо информацию из него вывести в шаблоне. Это делается так:

tm = Template("Мне {{p.name}} лет и зовут {{ p.age }}.")

msg = tm.render(p=per)

Мы здесь передаем ссылку на объект класса Person и метод render берет имя и возраст из этого объекта, используя ссылку p. Или, если в классе Person реализовать геттеры:

**def** getName(self):

**return** self.name

**def** getAge(self):

**return** self.age

То в шаблоне можно вызвать их для получения необходимой информации:

tm = Template("Мне {{p. getAge()}} лет и зовут {{ p.getName() }}.")

Это еще раз показывает гибкость двойных фигурных скобок, в которых можно записывать любые (разумные для шаблона) операторы языка Python.

Ну, и наконец, передавать данные в шаблон можно непосредственно с помощью словаря. Например:

per = { 'name': 'Федор', 'age': 34 }

tm = Template("Мне {{p.age}} лет и зовут {{ p.name }}.")

msg = tm.render(p=per)

Или, можно обратиться к данным по такому синтаксису:

tm = Template("Мне {{p['age']}} лет и зовут {{ p['name'] }}.")

Результат будет одинаковым. Вот так используется определение двойные фигурные скобки в модуле Jinja.

# Экранирование и блоки raw, for, if

[Смотреть материал на видео](https://www.youtube.com/watch?v=F63wc5nPdho&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/ekranirovanie-i-bloki-raw-for-if>

Продолжаем изучение модуля Jinja и вначале рассмотрим способы экранирования данных в строках. О чем здесь речь? Давайте представим, что мы бы хотели фрагмент шаблона никак не преобразовывать и выводить так, как он записан. Например, вот такую строку:

**from** jinja2 **import** Template

data = '''Модуль Jinja вместо

определения {{ name }}

подставляет соответствующее значение'''

tm = Template(data)

msg = tm.render(name='Федор')

**print**(msg)

Если выполнить эту программу, то на выходе получим строку:

Модуль Jinja вместо

определения Федор

подставляет соответствующее значение

Как вы понимаете, это не то, что мы бы хотели видеть. Нам нужно оставить эти фигурные скобки так, как они записаны без обработки. Как раз для этого используется специальное определение:

{% raw %} … {% endraw %}

И все записанное внутри этого блока не подвергается преобразованию модулем Jinja. То есть, если в строке data прописать этот блок:

data = '''{% raw %}Модуль Jinja вместо

определения {{ name }}

подставляет соответствующее значение{% endraw %}'''

То на выходе будет именно этот текст без каких-либо подстановок. Вот так работает блок raw.

## Экранирование символов

Далее, при работе с текстовыми HTML-шаблонами часто возникает необходимость экранирования некоторых символов, которые браузерами воспринимаются как определение тегов. Например, вот такая программа:

link = '''В HTML-документе ссылки определяются так:

<a href="#">Ссылка</a>'''

tm = Template("{{ link }}")

msg = tm.render(link = link)

**print**(msg)

На выходе дает строку:

В HTML-документе ссылки определяются так:

<a href="#">Ссылка</a>

И если, затем ее вывести как фрагмент HTML-документа, то мы увидим в окне браузера:

C:\Users\Andrey_User\Downloads\image001.png

А хотелось бы увидеть именно определение тега <a> вместо самой ссылки. Для этого внутри фигурных скобок можно прописывать различные флаги и один из них

e – escape (экранирование)

Применить его можно так:

tm = Template("{{ link | e }}")

Теперь, на выходе имеем:

В HTML-документе ссылки определяются так:

&lt;a href=&#34;#&#34;&gt;Ссылка&lt;/a&gt;

Смотрите, здесь вместо угловых скобок и кавычек прописаны специальные символы, которые браузером будут восприниматься как текстовый фрагмент и отображаться в виде:

C:\Users\Andrey_User\Downloads\image002.png

И это именно то, что мы бы хотели получить в данном случае. Подобное преобразование довольно часто требуется выполнять при работе с шаблонами страниц сайтов, поэтому в модуле Jinja существует специальный класс escape, который на выходе выдает строку с экранированными символами:

**from** jinja2 **import** Template, escape

link = '''В HTML-документе ссылки определяются так:

<a href="#">Ссылка</a>'''

**print**(escape(link))

Мы здесь уже не используем функционал шаблона, а просто преобразуем строку. Поэтому, для целей простого экранирования, лучше использовать этот метод, т.к. он работает быстрее, нежели через шаблон. Вот так работает экранирование в модуле Jinja.

## Выражение for

Следующий вид блока, который мы рассмотрим – это

{% for <выражение> -%}

    <повторяемый фрагмент>

{% endfor %}

Он позволяет формировать список на основе любого итерируемого объекта, например, упорядоченного списка. В качестве примера я приведу такую программу:

cities = [{'id': 1, 'city': 'Москва'},

          {'id': 5, 'city': 'Тверь'},

          {'id': 7, 'city': 'Минск'},

          {'id': 8, 'city': 'Смоленск'},

          {'id': 11, 'city': 'Калуга'}]

link = '''<select name="cities">

{% for c in cities %}

    <option value="{{c['id']}}">{{c['city']}}</option>

{% endfor %}

</select>'''

tm = Template(link)

msg = tm.render(cities = cities)

**print**(msg)

У нас здесь имеется список из городов и их id (например, в соответствии с rowid БД SQLite) и, затем, формируется тег select для HTML-документа. Внутри размещаются теги option по указанному формату для каждого города. В результате выполнения такого шаблона, получим следующее:

<select name="cities">

    <option value="1">Москва</option>

    <option value="5">Тверь</option>

Мы здесь видим лишний перенос строки между тегами. Это из-за того, что после блока for стоит перенос строки. Убрать его можно двумя способами. Или записать вот в таком виде:

link = '''<select name="cities">

{% for c in cities %}<option value="{{c['id']}}">{{c['city']}}</option>{% endfor %}

</select>'''

Или, поставить знак минус перед закрывающей скобкой:

link = '''<select name="cities">

{% for c in cities -%}

    <option value="{{c['id']}}">{{c['city']}}</option>

{% endfor -%}

</select>'''

Этот минус указывает удалять все переносы и пробелы после блока for и после endfor. Его можно ставить и вначале, тогда будут удалены пробелы и переносы справа от скобки.

## Выражение if

Последний тип блока, который мы рассмотрим на этом занятии – это блок для проверки условий. В самом простом варианте он записывается в виде:

{% if  <условие> %}

    <фрагмент при истинности условия>

{% endif %}

И в нашем примере его можно использовать так:

link = '''<select name="cities">

{% for c in cities -%}

{% if c.id > 6 -%}

    <option value="{{c['id']}}">{{c['city']}}</option>

{% endif -%}

{% endfor -%}

</select>'''

Мы здесь добавили проверку, что добавлять следует только те города, у которых id больше 6.

Соответственно, здесь можно добавлять определение else:

link = '''<select name="cities">

{% for c in cities -%}

{% if c.id > 6 -%}

    <option value="{{c['id']}}">{{c['city']}}</option>

{%else -%}

    {{c['city']}}

{% endif -%}

{% endfor -%}

</select>'''

И elif:

link = '''<select name="cities">

{% for c in cities -%}

{% if c.id > 6 -%}

    <option value="{{c['id']}}">{{c['city']}}</option>

{%elif c.city == "Москва" -%}

    <option>{{c['city']}}</option>

{%else -%}

    {{c['city']}}

{% endif -%}

{% endfor -%}

</select>'''

То есть, этот блок используется ровно так как и оператор if  в Python.

# Фильтры и макросы: macro, call

[Смотреть материал на видео](https://www.youtube.com/watch?v=dsUOZXM1GAM&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/filtry-i-makrosy-macro-call>

Продолжаем изучение модуля Jinja и подробнее рассмотрим фильтры, которые удобно применять для получения более сложных представлений. И первый часто применяемый фильтр – это sum для вычисления суммы определенной коллекции. Например, у нас имеется список автомобилей:

cars = [

    {'model': 'Ауди', 'price': 23000},

    {'model': 'Шкода', 'price': 17300},

    {'model': 'Вольво', 'price': 44300},

    {'model': 'Фольксваген', 'price': 21300}

]

Требуется вывести суммарную цену всех автомобилей. Для этого можно прописать такой шаблон:

tpl = "Суммарная цена автомобилей {{ cs | sum(attribute='price') }}"

tm = Template(tpl)

msg = tm.render(cs = cars)

**print**(msg)

Смотрите, мы здесь сначала указываем коллекцию cs, из которой следует брать атрибут price и суммировать его значения по всем автомобилям. В результате применения такого фильтра в фигурных скобках будет подставлено одно число, равное сумме всех полей price:

Суммарная цена автомобилей 105900

А вот если убрать этот фильтр:

tpl = "Суммарная цена автомобилей {{ cs  }}"

то увидим коллекцию автомобилей:

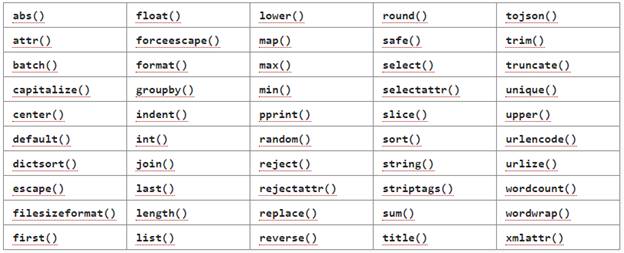
Суммарная цена автомобилей [{'model': 'Ауди', 'price': 23000}, {'model': 'Шкода', 'price': 17300}, {'model': 'Вольво', 'price': 44300}, {'model': 'Фольксваген', 'price': 21300}]

Вот так фильтр может кардинально менять поведение шаблона. В общем случае синтаксис фильтра sum, следующий:

sum(iterable, attribute=None, start=0)

Здесь последний параметр start – это прибавочное значение к вычисленной сумме.

Полный список доступных фильтров, следующий:



Они все вполне очевидны и подробное описание можно почитать на странице официальной документации:

<https://jinja.palletsprojects.com/en/2.11.x/templates/>

Я здесь приведу лишь еще несколько примеров:

tpl = "Автомобиль: {{ cs | max(attribute='price')  }}"

Выводит словарь для автомобиля с максимальной ценой:

Автомобиль: {'model': 'Вольво', 'price': 44300}

Для выбора отдельного поля следует использовать такую запись:

tpl = "Автомобиль: {{ (cs | max(attribute='price')).model  }}"

то есть, мы сначала получаем словарь, а затем, выбираем из него поле model.

То же самое будет и для фильтра min:

tpl = "Автомобиль: {{ (cs | min(attribute='price')).model  }}"

Выбор случайного значения из последовательности:

tpl = "Автомобиль: {{ cs | random  }}"

Замена малой буквы ‘о’ на заглавную:

tpl = 'Автомобиль: {{ cs | replace("о", "О") }}'

И так далее. Каждый из этих фильтров можно применять в соответствии с его описанием и на выходе будем получать соответствующие данные.

## Блок filter

Большую гибкость в применении фильтров дает специальный блок:

{{% filter <название фильтра> %}

<фрагмент для применения фильтра>

{% endfilter %}

Например, сделать так:

persons = [

    {"name": "Алексей", "old": 18, "weight": 78.5},

    {"name": "Николай", "old": 28, "weight": 82.3},

    {"name": "Иван", "old": 33, "weight": 94.0}

]

tpl = '''

{%- for u in users -%}

{% filter upper %}{{u.name}}{% endfilter %}

{% endfor -%}

'''

tm = Template(tpl)

msg = tm.render(users = persons)

**print**(msg)

На выходе получим список имен, записанных заглавными буквами.

## Макроопределения

Модуль Jinja поддерживает макроопределения для шаблонов, которые весьма полезны, чтобы избежать повторяемых определений в соответствии с принципом

DRY – Don’t Repeat Yourself (не повторяйся)

Например, нам необходимо создать несколько полей ввода input в шаблоне HTML-документа. Его можно задать так:

html = '''

{% macro input(name, value='', type='text', size=20) -%}

    <input type="{{ type }}" name="{{ name }}" value="{{ value|e }}" size="{{ size }}">

{%- endmacro %}

{{ input('username') }}

{{ input('email') }}

{{ input('password') }}

'''

Здесь с помощью ключевого слова macro задано макроопределение с именем input и набором параметров. Это очень похоже на определение функций в Python. Учитывая, что в качестве параметров можно указывать специальные:

* varargs – список переданных значений (параметров);
* kwargs – список переданных именованных параметров.

А далее, мы используем этот макрос для создания трех полей input. Как видите, это может быть очень удобно. По аналогии можно создавать другие макросы и, затем, многократно их использовать в шаблонах.

## Вложенные макросы – call

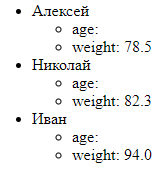
Модуль Jinja имеет специальное определение:

{% call[(параметры)] <вызов макроса> %}

<вложенный шаблон>

{% endcall %}

которое позволяет создавать своего рода вложенные макросы. Проще всего понять работу этого блока на конкретном примере. Предположим, мы хотим сформировать вот такой список:



На уровне HTML-документа это выглядит так:

<ul>

<li>Алексей

    <ul>

    <li>age:

    <li>weight: 78.5

    </ul>

<li>Николай

    <ul>

    <li>age:

    <li>weight: 82.3

    </ul>

<li>Иван

    <ul>

    <li>age:

    <li>weight: 94.0

    </ul>

</ul>

Создадим шаблон, который позволяет генерировать такой список на основе коллекции:

persons = [

    {"name": "Алексей", "old": 18, "weight": 78.5},

    {"name": "Николай", "old": 28, "weight": 82.3},

    {"name": "Иван", "old": 33, "weight": 94.0}

]

Далее, определим макрос, который генерирует начальный список из имен. Его можно представить так:

html = '''

{% macro list\_users(list\_of\_user) -%}

<ul>

{% for u in users -%}

    <li>{{u.name}}

{%- endfor %}

</ul>

{%- endmacro %}

{{list\_users(users)}}

'''

tm = Template(html)

msg = tm.render(users = persons)

**print**(msg)

После выполнения этой программы, увидим следующее:

<ul>

<li>Алексей<li>Николай<li>Иван

</ul>

Отлично, это сделали. А теперь для каждого человека добавим вложенный список с его возрастом и весом. И сделаем это через блок call. Определим его так:

html = '''

{% macro list\_users(list\_of\_user) -%}

<ul>

{% for u in list\_of\_user -%}

    <li>{{u.name}} {{caller(u)}}

{%- endfor %}

</ul>

{%- endmacro %}

{% call(user) list\_users(users) %}

    <ul>

    <li>age: {{user.old}}

    <li>weight: {{user.weight}}

    </ul>

{% endcall -%}

'''

Смотрите, мы здесь прописали call с передаваемым ему параметром user – это будет текущий словарь, взятый из списка persons. Далее, указываем макрос, который следует вызвать для этого блока call. А все что записано внутри этого блока будет подставлено на место вызова метода caller внутри макроса list\_users. В результате будет сформирован искомый список.

# Загрузчики: FileSystemLoader, PackageLoader, FunctionLoader и другие

[Смотреть материал на видео](https://www.youtube.com/watch?v=1Dsy2dKNP3Y&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/zagruzchiki-filesystemloader-packageloader-functionloader>

Рассматриваемые шаблоны на предыдущих занятиях были записаны в виде многострочного текста, но на практике, шаблоны, как правило, хранятся в отдельных текстовых файлах и загружаются по мере необходимости. Для этого в Jinja реализован класс Enviroment (окружение), который, наверное, представляет собой некий центральный объект, через который и происходит основная работа с API данного пакета. Я здесь не буду приводить подробное описание параметров этого класса, так как это сугубо справочная информация и ее можно самостоятельно изучить вот по этой ссылке:

<https://jinja.palletsprojects.com/en/2.11.x/api/#jinja2.Environment>

Вместо этого мы рассмотрим примеры использования этого объекта. Итак, предположим, что все наши шаблоны хранятся в подкаталоге

templates

относительно рабочего каталога программы. В частности, там находится файл main.htm со следующим содержимым:

<!DOCTYPE html>

<html>

<head>

         <base href="https://proproprogs.ru/">

         <title>Про программирование</title>

</head>

<body>

<ul>

{% **for** u **in** users -%}

    <li>{{u.name}}

{% endfor -%}

</ul>

</body>

</html>

Здесь в html документе находится фрагмент шаблона, который будет в дальнейшем нами применен. Сама же программа, имеет вид:

**from** jinja2 **import** Environment, FileSystemLoader

persons = [

    {"name": "Алексей", "old": 18, "weight": 78.5},

    {"name": "Николай", "old": 28, "weight": 82.3},

    {"name": "Иван", "old": 33, "weight": 94.0}

]

file\_loader = FileSystemLoader('templates')

env = Environment(loader=file\_loader)

tm = env.get\_template('main.htm')

msg = tm.render(users = persons)

**print**(msg)

Мы здесь вначале импортируем модули Environment и FileSystemLoader, затем, создаем файловый загрузчик, который будет брать шаблоны из подкаталога 'templates'. Далее, создается объект Environment, у которого параметр loader ссылается на файловый загрузчик. В результате, мы можем загружать и обрабатывать любые шаблоны, находящиеся в подкаталоге 'templates'. С помощью метода get\_template загружается файл 'main.htm' и возвращается объект шаблона (Template) с его содержимым. В конце вызывается метод render для обработки шаблона и формирования строки HTML-документа на основе списка persons.

Обратите внимание, по умолчанию Python работает со строками в формате Юникода, поэтому для корректного представления кириллицы необходимо сохранять текстовые файлы в формате UTF8.

Помимо FileSystemLoader, который отвечает за загрузку шаблонов непосредственно из файловой системы, в Jinja есть еще несколько предопределенных загрузчиков:

* PackageLoader – для загрузки шаблонов из пакета;
* DictLoader – для загрузки шаблонов из словаря;
* FunctionLoader – для загрузки на основе функции;
* PrefixLoader – загрузчик, использующий словарь для построения подкаталогов;
* ChoiceLoader – загрузчик, содержащий список других загрузчиков (если один не сработает, выбирается следующий);
* ModuleLoader – загрузчик для скомпилированных шаблонов.

Например, FunctionLoader можно реализовать так. Определим некую функцию, которая будет возвращать шаблон:

**def** loadTpl(path):

**if** path == "index":

**return** '''Имя {{u.name}}, возраст {{u.old}}'''

**else**:

**return** '''Данные: {{u}}'''

И, затем, реализуем загрузчик на ее основе:

file\_loader = FunctionLoader(loadTpl)

env = Environment(loader=file\_loader)

tm = env.get\_template('index')

msg = tm.render(u = persons[0])

Так как берется шаблон 'index', то именно это значение будет принимать параметр path и функция возвратит первый шаблон. Если указать любое другое значение, то получим второй шаблон.

Использование остальных загрузчиков выполняется похожим образом, подробное описание, как всегда, можно посмотреть на странице официальной документации:

<https://jinja.palletsprojects.com/en/2.11.x/api/#jinja2.FileSystemLoader>

# Конструкции include и import

[Смотреть материал на видео](https://www.youtube.com/watch?v=Ree-JFi06y8&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/konstrukcii-include-i-import>

Часто при создании сайтов страницу делят, как минимум на три части:

заголовок, контент, футер

Это связано с тем, что у всех страниц шапка (заголовок) сайта, как правило, одинакова и потому ее нужно просто подключить к нужному шаблону. То же самое и с футером (подвалом) страницы. В рамках модуля Jnija это можно реализовать с помощью специального блока

{% include <путь к файлу шаблона> %}

Например, сделать так. Разобьем страницу main.htm на три блока:

**- заголовок (header.htm):**

<!DOCTYPE html>

<html>

<head>

         <base href="https://proproprogs.ru/">

         <title>Про программирование</title>

</head>

<body>

**- футер (footer.htm):**

</body>

</html>

**- страницу (page.htm)**

{% include 'header.htm' %}

Содержимое страницы

{% include 'footer.htm' %}

Теперь, в программе можно просто загрузить шаблон page.htm и получить целиком собранную HTML-страницу:

file\_loader = FileSystemLoader('templates')

env = Environment(loader=file\_loader)

tm = env.get\_template('page.htm')

msg = tm.render()

**print**(msg)

Блок include можно записывать с дополнительными параметрами:

{% include "header.html" ignore missing %}

В этом случае отсутствие шаблона не будет приводить к появлению исключения TemplateNotFound. Например, записав вот такую строчку:

{% include 'header.html' ignore missing %}

получим шаблон без 'header.html', т.к. такой файл отсутствует в текущем каталоге.

Вернем запись:

{% include 'header.htm' ignore missing %}

а в самом файле 'header.htm' пропишем шаблон для домена и заголовка страницы:

<!DOCTYPE html>

<html>

<head>

         <meta charset="UTF-8">

         <base href="{{domain}}">

         <title>{{title}}</title>

</head>

<body>

Теперь, вызывая шаблон 'page.htm' можем передавать ему домен и заголовок для каждой страницы:

tm = env.get\_template('page.htm')

msg = tm.render(domain='http://proproprogs.ru', title="Про Jinja")

Этот пример наглядно показывает, что шаблон сначала собирается, а затем, итоговая страница наполняется конкретным содержимым, то есть, render обрабатывает ее целиком.

Если в блоке include требуется подключить сразу несколько файлов, то их следует указать в виде списка:

{% include ['page1.htm', 'page2.htm'] ignore missing %}

## Конструкция import

Модуль Jinja позволяет не только включать отдельные файлы в общий шаблон, но и импортировать их. Отличие в том, что при импорте шаблон не добавляется, но можно использовать его отдельные конструкции. Например, представим себе, что в HTML-документе могут присутствовать диалоговые окна. Их реализацию опишем в виде макроса, который расположим в отдельном файле dialogs.htm:

{% macro dialog\_1(title, msg='') -%}

<div **class**="dialog">

<p **class**="title">{{title}}</p>

<p **class**="message">{{msg}}</p>

<input type="button" value="Закрыть"></p>

</div>

{%- endmacro %}

И, затем, импортируем его в основном шаблоне page.htm:

{% include 'header.htm' ignore missing %}

{% **import** 'dialogs.htm' **as** dlg %}

Содержимое страницы

{{ dlg.dialog\_1('Внимание', 'Это тестовый диалог') }}

{% include 'footer.htm' %}

Смотрите, благодаря импорту мы можем обратиться к макросу dialog\_1 и создать вид диалогового окна. Причем, во избежание конфликтов имен, в конце после ключевого слова as можно прописать алиас, через который будет происходить обращение к элементам шаблона dialogs.htm.

Или же, воспользоваться такой конструкцией:

{% include 'header.htm' ignore missing  %}

{% **from** 'dialogs.htm' **import** dialog\_1 **as** dlg %}

Содержимое страницы

{{ dlg('Внимание', 'Это тестовый диалог') }}

{% include 'footer.htm' %}

Тогда обращение к макросу будет происходить по имени dlg.

# Конструкции include и import

[Смотреть материал на видео](https://www.youtube.com/watch?v=Ree-JFi06y8&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

<https://proproprogs.ru/modules/konstrukcii-include-i-import>

Часто при создании сайтов страницу делят, как минимум на три части:

заголовок, контент, футер

Это связано с тем, что у всех страниц шапка (заголовок) сайта, как правило, одинакова и потому ее нужно просто подключить к нужному шаблону. То же самое и с футером (подвалом) страницы. В рамках модуля Jnija это можно реализовать с помощью специального блока

{% include <путь к файлу шаблона> %}

Например, сделать так. Разобьем страницу main.htm на три блока:

**- заголовок (header.htm):**

<!DOCTYPE html>

<html>

<head>

         <base href="https://proproprogs.ru/">

         <title>Про программирование</title>

</head>

<body>

**- футер (footer.htm):**

</body>

</html>

**- страницу (page.htm)**

{% include 'header.htm' %}

Содержимое страницы

{% include 'footer.htm' %}

Теперь, в программе можно просто загрузить шаблон page.htm и получить целиком собранную HTML-страницу:

file\_loader = FileSystemLoader('templates')

env = Environment(loader=file\_loader)

tm = env.get\_template('page.htm')

msg = tm.render()

**print**(msg)

Блок include можно записывать с дополнительными параметрами:

{% include "header.html" ignore missing %}

В этом случае отсутствие шаблона не будет приводить к появлению исключения TemplateNotFound. Например, записав вот такую строчку:

{% include 'header.html' ignore missing %}

получим шаблон без 'header.html', т.к. такой файл отсутствует в текущем каталоге.

Вернем запись:

{% include 'header.htm' ignore missing %}

а в самом файле 'header.htm' пропишем шаблон для домена и заголовка страницы:

<!DOCTYPE html>

<html>

<head>

         <meta charset="UTF-8">

         <base href="{{domain}}">

         <title>{{title}}</title>

</head>

<body>

Теперь, вызывая шаблон 'page.htm' можем передавать ему домен и заголовок для каждой страницы:

tm = env.get\_template('page.htm')

msg = tm.render(domain='http://proproprogs.ru', title="Про Jinja")

Этот пример наглядно показывает, что шаблон сначала собирается, а затем, итоговая страница наполняется конкретным содержимым, то есть, render обрабатывает ее целиком.

Если в блоке include требуется подключить сразу несколько файлов, то их следует указать в виде списка:

{% include ['page1.htm', 'page2.htm'] ignore missing %}

## Конструкция import

Модуль Jinja позволяет не только включать отдельные файлы в общий шаблон, но и импортировать их. Отличие в том, что при импорте шаблон не добавляется, но можно использовать его отдельные конструкции. Например, представим себе, что в HTML-документе могут присутствовать диалоговые окна. Их реализацию опишем в виде макроса, который расположим в отдельном файле dialogs.htm:

{% macro dialog\_1(title, msg='') -%}

<div **class**="dialog">

<p **class**="title">{{title}}</p>

<p **class**="message">{{msg}}</p>

<input type="button" value="Закрыть"></p>

</div>

{%- endmacro %}

И, затем, импортируем его в основном шаблоне page.htm:

{% include 'header.htm' ignore missing %}

{% **import** 'dialogs.htm' **as** dlg %}

Содержимое страницы

{{ dlg.dialog\_1('Внимание', 'Это тестовый диалог') }}

{% include 'footer.htm' %}

Смотрите, благодаря импорту мы можем обратиться к макросу dialog\_1 и создать вид диалогового окна. Причем, во избежание конфликтов имен, в конце после ключевого слова as можно прописать алиас, через который будет происходить обращение к элементам шаблона dialogs.htm.

Или же, воспользоваться такой конструкцией:

{% include 'header.htm' ignore missing  %}

{% **from** 'dialogs.htm' **import** dialog\_1 **as** dlg %}

Содержимое страницы

{{ dlg('Внимание', 'Это тестовый диалог') }}

{% include 'footer.htm' %}

Тогда обращение к макросу будет происходить по имени dlg.

# Наследование расширение шаблонов

[Смотреть материал на видео](https://www.youtube.com/watch?v=RYIGMcJMy6g&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U" \t "_blank)

На этом занятии поговорим о механизме расширения шаблонов в Jninja. Его еще называют наследованием. Это достаточно удобный инструмент, уменьшающий объем дублируемого кода в шаблонах. Рассмотрим принцип его работы на нашем примере с HTML-страницами сайта.

Для простоты восприятия возьмем целостную страницу и представим ее в таком виде (файл ex\_main.htm):

<!DOCTYPE html>

<html>

<head>

         <meta charset="UTF-8">

         <title>{% block title %}{% endblock %}</title>

</head>

<body>

{% block content %}

{% endblock %}

</body>

</html>

Смотрите, здесь используется новый тип блоков – именованные блоки, которые в самом простом случае записываются по синтаксису:

{% block <имя блока> %}  
{% endblock %}

Эти блоки как раз и используются для создания расширения базового шаблона страницы ex\_main.htm для создания всех страниц текущего сайта. Расширение (или наследование) шаблона делается следующим образом (файл about.htm):

{% extends 'ex\_main.htm' %}

{% block title%}О сайте{% endblock %}

{% block content %}

<h1>О сайте</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

Первой строчкой мы указываем базовый шаблон 'ex\_main.htm', который собираемся расширять в шаблоне about.htm. Затем, указываем, что первый именованный блок title будет содержать строку «О сайте», а второй (content) – текст, помещенный в него.

Далее, в самой программе на Python мы делаем следующее:

**from** jinja2 **import** Environment, FileSystemLoader

file\_loader = FileSystemLoader('templates')

env = Environment(loader=file\_loader)

template = env.get\_template('about.htm')

output = template.render()

**print**(output)

Используя файловый загрузчик, берем файл шаблона about.htm из подкаталога templates и, затем, обрабатываем его с помощью метода render. На выходе получим следующую HTML-страницу:

<!DOCTYPE html>

<html>

<head>

         <meta charset="UTF-8">

         <title>О сайте</title>

</head>

<body>

<h1>О сайте</h1>

Классный сайт, если его доделать.</p>

</body>

</html>

Как видите, благодаря использованию механизма наследования шаблонов, легко и просто можно формировать страницы самого разного уровня сложности.

Разумеется, если базовый шаблон находится в другом каталоге относительно дочернего about.htm, то в инструкции extends это явно нужно прописать, например, так:

{% extends 'layout/default.tpl' %}

Будет взят шаблон default.tpl из подкаталога layout, находящийся в каталоге templates.

Далее, если нам нужно один и тот же блок использовать несколько раз на странице, то это делается так:

{% extends 'layout/default.tpl' %}

{% block title%}О сайте{% endblock %}

{% block content %}

<h1>{{ self.title() }}</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

Здесь self ссылается на объект текущего шаблона, в котором имеется метод title, т.к. существует блок с таким именем. И, далее, вызывая его, будет напечатано содержимое этого блока. Так можно делать с любым именованным блоком.

Ну а раз есть параметр self, то должен быть и super, который обращается к блоку базового шаблона и берет информацию непосредственно из него. В качестве простой демонстрации добавим в блок content базового шаблона строку «Блок контента». А в дочернем шаблоне about.htm сделаем вызов:

{% block content %}

{{ super() }}

<h1>{{ self.title() }}</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

В итоге, получим результат:

Блок контента  
<h1>О сайте</h1>  
<p>Классный сайт, если его доделать.</p>

Если же вызов super убрать, то останутся только две строчки:

<h1>О сайте</h1>  
<p>Классный сайт, если его доделать.</p>

Этот пример также показывает, что содержимое именованных блоков полностью замещается содержимым, указанным в дочерних шаблонах. И если нам нужно добавить новую информацию в производном шаблоне к информации блока базового шаблона, то следует вызывать метод super.

## Вложенные блоки

При необходимости блоки можно вкладывать друг в друга создавая их иерархию. Например, пропишем в шаблоне базового класса в блоке content еще один блок – table\_contents:

{% block content %}

         {% block table\_contents %}

         <ul>

         {% **for** li **in** list\_table -%}

         <li>{{li}}</li>

         {% endfor -%}

         </ul>

         {% endblock table\_contents %}

{% endblock content %}

Обратите внимание, здесь после endblock дополнительно указано какой блок заканчивается. Это необязательная запись, но она помогает лучше ориентироваться в сложных шаблонах. Далее, в дочернем шаблоне about.htm обратиться к этому вложенному блоку можно так:

{% block content %}

{{ super() }}

<h1>{{ self.title() }}</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

Или, так:

{% block content %}

{% block table\_contents %}{{ super() }}{% endblock %}

<h1>{{ self.title() }}</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

Вот этот второй вариант более гибкий, т.к. в block content базового шаблона могут присутствовать и другие именованные блоки и мы здесь добавляем только один - table\_contents, остальные будут проигнорированы. Если же убрать строчку с вложенным блоком:

{% block content %}

<h1>{{ self.title() }}</h1>

Классный сайт, если его доделать.</p>

{% endblock %}

То получим страницу без оглавления. Видите, это очень удобно: если нужно добавляем буквально одной строчкой, а если не надо, то ничего не пишем.

## Область видимости блоков

Давайте теперь, немного усовершенствуем базовый шаблон и добавим еще один блок для формирования элементов списка:

{% **for** li **in** list\_table -%}

<li>{% block item %}{{ li }}{% endblock %}</li>

{% endfor -%}

Если теперь выполнить программу, то внутри тегов li не будет никакой информации. Дело в том, что внутри блока item доступ к внешней переменной li нет. Чтобы исправить эту ситуацию и разрешить оперировать переменными из внешней области видимости, после имени блока следует прописать ключевое слово scoped:

<li>{% block item scoped %}{{ li }}{% endblock %}</li>

Теперь при запуске программа будет работать также, как и ранее. Но мы же добавили этот блок item не просто так, значит, собираемся его переопределять в дочернем шаблоне. И это можно сделать следующим образом:

{% block item %}<p **class**="item">{{ super() }}</p>{% endblock %}

Смотрите, мы здесь воспользовались функцией super, чтобы получить текущее значение списка и дополнительно еще прописали тег p с соответствующим стилем оформления. При запуске увидим следующий результат:

<ul>

<li><p class="item">Математика</p></li>

<li><p class="item">Физика</p></li>

<li><p class="item">Русский</p></li>

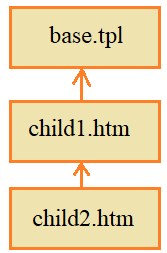
<li><p class="item">Информатика</p></li>

</ul>

Вот такая гибкость может быть достигнута за счет использования механизма наследования и именованных блоков.

## Вложенное наследование шаблонов

В заключение этого занятия отмечу один, в общем-то, очевидный момент: шаблоны поддерживают вложенное наследование, то есть, формирование итогового дочернего шаблона по цепочке: от базового (корневого) до последнего – выходного. Например, вот такую иерархию:



Можно организовать так:

* файл base.tpl – такой же как и ex\_main.htm:
* файл child1.htm: {% extends 'base.tpl' %} …
* файл child2.htm: {% extends 'child1.htm' %} …

И их дальнейшая обработка происходит аналогичным образом.

На этих занятиях мы с вами рассмотрели основные, базовые возможности пакета Jinja. Конечно, это не все возможности, которыми он обладает. Например, мы совершенно не касались темы расширения базовых классов пакета для создания более тонкого рендеринга шаблонов, или реализации своего собственного загрузчика. Но это уже тонкости, которые требуются далеко не в каждом проекте. В большинстве случаев рассмотренных базовых операций Jinja вполне достаточно для использования шаблонов разного уровня сложности в своих проектах. Кроме того, этот материал поможет лучше понимать принцип работы таких популярных фреймворков как Flask или Django, которые используют ту же концепцию шаблонов для страниц сайтов. Ну а для тех, кто хочет углубиться в эту тему, могу посоветовать посмотреть официальную документацию по пакету Jinja:

<https://jinja.palletsprojects.com/en/2.11.x>