PyLadies

Vienna 13.11.2020

Who?

International mentorship group with a focus on helping more women become active participants and leaders in the Python open-source community.

Our mission is to promote, educate and advance a diverse Python community through outreach, education, conferences, events and social gatherings.

Agenda for today

---

1. Advanced string formatting
2. Templating using Jinja2
3. Iterators, Generators
4. Place for all your questions

Goals

- Learn new data structures, methods and functions
- Understand templating
- Work on your own project

Advanced string formatting

---

1. “old” string formatting
    - print('The value of pi is approximately %5.3f.' % math.pi)
    - using % (modulo) operator
    - options can be found: https://docs.python.org/3/library/stdtypes.html#old-string-formatting
    - Troubles printing tuples and dictionaries, not preferred way anymore

Advanced string formatting

---

2. Formatted String Literals

- **print(f'The value of pi is approximately {math.pi:.3f}.')**
- string if prefixed with f (or F) and expression is put into {}
- Options for converting value first: **!a** (to ascii()), **!s** (to str()), **!r** (to repr())
- Easy to align values:
- **table = {'Tyna': 123456, 'Lubo': 758964, 'Sylvia': 547896}**
- **for name, phone_number in table.items():**
- **   print(f'{name:10} ==> {phone_number:10d}')**

Advanced string formatting

---

3. format() method

- **print('We are the {} who say "{}!"'.format('knights', 'Ni'))**
- Lots of option:
- {1} - referencing string on a second position
- {name} - referencing string with a same name in **format(name='Tyna')**
- Can be mixed together, and also used for aligned table
- **for x in range(1, 11):**
- **   print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))**

Advanced string formatting

---

4. Manual string formatting

- **for x in range(1, 11):**
- ** print(repr(x).rjust(2), repr(x*x).rjust(3))**
- **str.rjust()** - right-justifies a string, given a padding as “”
- also **str.ljust()** and **str.center()**
- **str.zfill()** - pads a numeric string on the left with zeros
- can lead into troubles with unexpected results - no truncating

Exercise

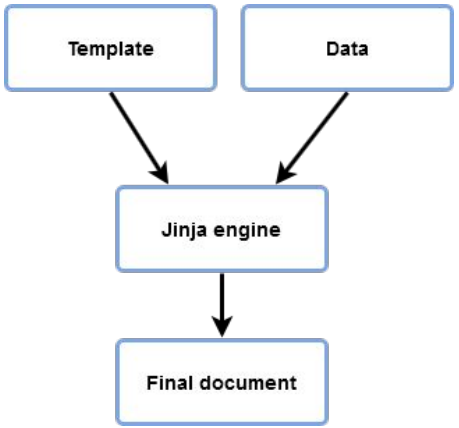
- Given a 5 element tuple: (4, 30, 2017, 2, 27), use string formatting to print: **'02 27 2017 04 30'**
- Given the following four element list: **['oranges', 1.3, 'lemons', 1.1]**. Write an f-string that will display: The weight of an orange is 1.3 and the weight of a lemon is 1.1. Try to use all possible methods
- Write a format string that will take the following four element tuple: (2, 123.4567, 10000, 12345.67) and produce: **'file_002, 123.46, 1.00e+04, 1.23e+04'**

Templates

- Template itself looks usually similar to the output. It has placeholders, common style and visual elements.
- Data to be inserted and data needed for template itself
- When there are ‘Logic or control-flow’ statements inside template, then output can look quite different
- Most common in web world (HTML, XML), but options are limitless - YML, Word, PDF …
- Why we talk about it? Django/Flask workshops

Jinja2

- Modern templating for Python, modelled after Django
- Control structures (loops, conditions)
- Inbuilt Filters and tests, macros, good customization
- HTML escaping preventing XSS Attack, sandbox environment for rendering untrusted templates
- Template inheritance
- data as python dict, json, yml



Jinja2 template examples

- {%....%} - for statements
- {{....}} - expressions used to print to template output
- {#....#} - comments which are not included in the output
- variable rendered as string: **{{ name }}**
- **undefined** rendered as empty string, unless **StrictUndefined** used
- loops on array with dicts:

```
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
```

- you can access keys using **dot** notation if they do not have **dot** in the name (otherwise Python [] subscript notation)

Jinja2 how to render

```
from jinja2 import Template
template = """hostname {{ hostname }}
ip name-server {{ server_primary }}"""
data = {
    "hostname": "pyladies",
    "server_primary": "1.1.1.1",
    "server_secondary": "8.8.8.8",
}
j2_template = Template(template)
print(j2_template.render(data)) # or write to file (with open(output.txt) as f: ...)
##########################################
# or load templates from folder and configure environment on app initialization
from jinja2 import Environment, PackageLoader, select_autoescape
env = Environment(
    loader=PackageLoader('yourapplication', 'templates'),
    autoescape=select_autoescape(['html', 'xml'])
)
```

Jinja2 template designer features

- chainable built-in **filters** done by **pipe |** symbol
  {{name|striptags|title}} is an equivalent of Pythonic:
  title(striptags(name))
- **tests** - {% if loop.index is divisibleby(3) %}
- **whitespace control** (newline default) **-** sign removes it from loop

```
{% for item in seq -%}
    {{ item }}
{%- endfor %}
```

- raw text not substituted inside template {% raw %} {% endraw %}
- child can overide template content from parent defined as
  {% block content-holder-1 %}{% endblock %}
- if-else conditions:

```
{% if kenny.sick %} Kenny is sick.
{% elif kenny.dead %} You killed Kenny!  You bastard!!!
{% else %} Kenny looks okay --- so far
{% endif %}
```

Jinja2 small project to practice

- Simple html webpage shown in flask app with examples of:
- string, loop, if/else, 1 block element filled by child template, comments
- we put simple starter files on github in case you need them:
- https://github.com/UndeadFairy/pyladies_vienna/tree/master/advanced_python/
- https://jinja.palletsprojects.com/en/2.11.x/templates/

Iterators

Iterators X Iterables

- Iterables: strings, tuples, lists…
- Iterator is an object you can iterate over all values
- in Python: iterator must have two methods - **__iter__()** and **__next__()** to satisfy iterator protocol

Iterators

```
my_tuple = ("apple", "banana", "cherry")
my_item = iter(my_tuple)
print(next(my_item))
print(next(my_item))
print(next(my_item))

for item in my_tuple:
  print(item)
```

Your own iterator

- You can design your own class/object to be iterator
- To do that, you need to define **__iter__()** and **__next__()** methods
- **__iter__()** must return self object itself but can do operations in between
- **__next__()** also allows operation but must return next item in given sequence

Your own iterator

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

- can prevent infinite loops
- Previous example would run forever if you would have enough next() statements
- StopIteration can determine when to stop

Generators

Consist of two parts:

1. generator function
2. generator object

Generator function:

- instead of return is using **yield**
- whenever function use in a body **yield**, it is automatically becoming generator

Generators

```
def generator_function():
    yield 1
    yield 2
    yield 3
```

Generator objects:

- Returned by generator function. Generator objects are used either by calling the next method on the generator object or using the generator object in a for loop

Generators

———

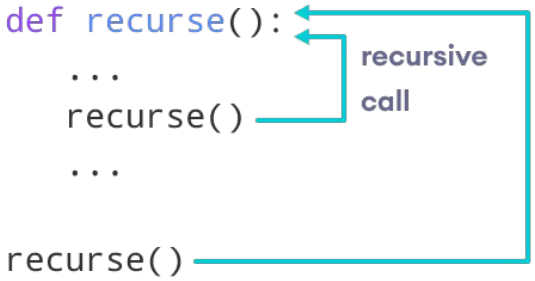
```
x = generator_function()
print(x.__next__())
print(x.__next__())
print(x.__next__())
```

So a generator function returns a generator object that is iterable → can be used as an Iterator

Exercise

- Write Fibonacci number sequence generator and use it. You can add some limitations not to go into infinite

Recursion

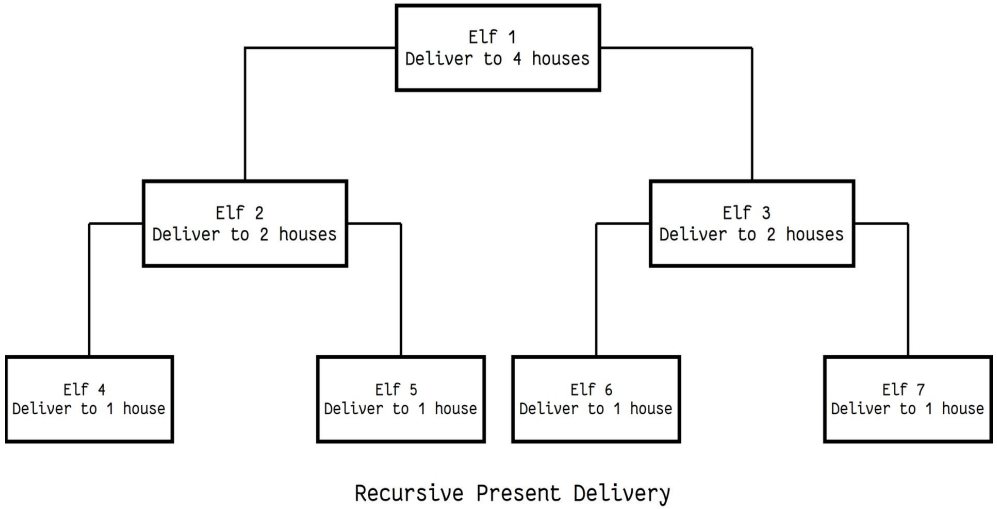


- Recursion is awesome, if you do not make it a never-stopping gluttonous monster, that eats up computing power in an instant with the best intent ^^
- Function that calls itself and running until stopped under some condition **if size == 1:**, **if task.is_done():**
- Example: factorial - 5! = 5*4*3*2*1 = 120

```
def factorial(x):
    """Recursive function to find the factorial of an integer"""
    if x == 1: # stopping the iteration too
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print(num, factorial(num))
```

Recursion exercise

- Santa delivers packages
- to children with help of
- his manager/worker elfs



Recursive Present Delivery

- Appoint elf and give all work to him
- Assign titles and responsibilities to the elves based on the number of houses for which they are responsible:
  - > 1 He is a manager and appoints two elves, divides his work to them
  - = 1 He is a worker and has to deliver the presents to the house

Resources and materials general

- advent of code - adventofcode.com
- hackerrank - hackerrank.com
- Django Girls django tutorial
- https://www.practicepython.org
- Nice Python exercices at one place https://github.com/tystar86/python_exercises
- https://automatetheboringstuff.com
- https://diveintopython3.problemsolving.io
- https://naucse.python.cz/2018/pyladies-en-prague/

Next topics

Graphics

GUI

Webscraping

fill the form regarding your interests please :) → https://forms.gle/UtfgVGe6AhhRwx539

Thank you and see you next time

Coding session - **26.11.2020** - 6PM - 8PM Online

Next workshop - **16.1.2021** - Web Scraping with Python