Published in Towards Data Science

Tomer Gabay Follow
Jan 16 · 6 min read · ✦ · Listen

5 Python Tricks That Distinguish Senior Developers From Juniors

Illustrated through differences in approaches to Advent of Code puzzles

Photo by Afif Ramdhasuma on Unsplash

Every year since 2015 on the first of December Advent of Code starts. As described on their website, Advent of Code (henceforth AoC) is

> *an Advent calendar of small programming puzzles for a variety of skill sets and skill levels that can be solved in any programming language you like. People use them as interview*

> *prep, company training, university coursework, practice problems, a speed contest, or to challenge each other.*

In this article, we'll take a look at five approaches to tackle common coding problems in a senior way instead of a junior one. Each coding problem is derived from an AoC puzzle, with many problems recurring multiple times throughout AoC and other coding challenges and assessments you might encounter e.g. in job interviews.

To illustrate the concepts I won't go into the solution of the full AoC puzzles, but rather only focus on a small part of a specific puzzle in which senior developers are easily distinguishable from juniors.

1. Read in a file effectively with comprehensions and splits

On Day1 of AoC it is required to read in several blocks of numbers. Each block is separated by an empty line (thus actually `'\n'`).

Input and desired output

```
# INPUT
10
20
30

50
60
70

# DESIRED OUTPUT
[[10, 20, 30], [50, 60 70]]
```

Junior developer approach: a loop with if-else statements

```
numbers = []
with open("file.txt") as f:
  group = []
  for line in f:
    if line == "\n":
      numbers.append(group)
      group = []
    else:
      group.append(int(line.rstrip()))
  # append the last group because if line == "\n" will not be True for
  # the last group
  numbers.append(group)
```

Senior developer approach: make use of list comprehensions and `.split()`

```
with open("file.txt") as f:
  nums = [list(map(int, (line.split()))) for line in f.read().rstrip().split("
```

Using list comprehensions we can fit the nine previous lines into one, without losing significant understandability or readability, and while gaining in performance (list comprehensions are faster than regular loops). For those that haven't seen `map` before, `map` maps a function (the first argument) to an iterable in the second argument. In this specific situation, it applies `int()` to every value in the list, making every item an integer. For more info about `map` click here.

2. Use Enum instead of if-elif-else

On Day2 the challenge revolves around a game of *rock-paper-scissors.* A different chosen shape (rock, paper, or scissors) results in a different amount of points: 1 (*X*), 2 (*Y*), and 3 (*Z*) respectively. Here below are two approaches to tackle this problem.

Input and desired output

```
# INPUT
X
Y
Z

# DESIRED OUTPUT
1
2
3
```

Junior developer approach: if-elif-else

```
def points_per_shape(shape: str) -> int:
  if shape == 'X':
    return 1
  elif shape == 'Y':
    return 2
  elif shape == 'Z':
    return 3
  else:
    raise ValueError('Invalid shape')
```

Senior developer approach: `Enum`

```
from enum import Enum

class ShapePoints(Enum):
  X = 1
  Y = 2
  Z = 3

def points_per_shape(shape: str) -> int:
  return ShapePoints[shape].value
```

Of course, in this example, the naive approach isn't *that* terrible but using `Enum` results in shorter and more readable code. Especially when more options are possible the naive *if-elif-else* approach will get worse and worse, while with `Enum` it stays relatively easy to keep an overview. For more on `Enum` click here.

3. Use lookup tables instead of dictionaries

In Day3 letters have different values. Lowercase a-z has values 1 through 26, and uppercase a-z has values 27 through 52. Because of the many different possible values, using `Enum` like here above would result in many lines of code. A more practical approach here is to use a *lookup table:*

```
# INPUT
c
```

```
Z
a
...

# DESIRED OUPUT
3
52
1
...
```

Junior developer approach: creating a global dictionary

```
letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
letter_dict = dict()
for value, letter in enumerate(letters, start=1):
  letter_dict[letter] = value

def letter_value(ltr: str) -> int:
  return letter_dict[ltr]
```

Senior developer approach: using a string as a lookup table

```
def letter_value(ltr: str) -> int
  return 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.index(ltr) + 1
```

Using the `.index()` method of a string we get the index, hence `letters.index('c')+1` will result in the expected value of 3. There is no need to store the values in a dictionary because the index *is* the value. To prevent the `+1` you could simply add a whitespace character at the beginning of the string so that the index of `a` starts on 1. However, this depends on whether you'd like to return a value of 0 for a whitespace or an error.

As you might have thought by now, yes, we could also solve the *rock, paper scissors* task using a lookup table:

```
def points_per_shape(shape: str) -> int:
  return 'XYZ'.index(shape) + 1
```

4. Advanced slicing

On Day5 it is required to read letters from lines (see input below). Each letter is on a fourth index, starting from index 1. Now, virtually every Python programmer will be familiar with string and list slicing using e.g. `list_[10:20]` . But what many people don't know is that you can define step size using e.g. `list_[10:20:2]` to define a step size of 2. On Day5 (and in many other coding situations) this could save you a lot of unnecessarily complicated code:

```
# INPUT
    [D]
[N] [C]
[Z] [M] [P]

# DESIRED OUTPUT
[' D ', 'NC', 'ZMP']
```

Junior developer approach: double for loop with `range` and indices

```
letters = []
with open('input.txt') as f:
  for line in f:
    row = ''
    for index in range(1, len(line), 4):
      row += line[index]
    letters.append(row)
```

Senior developer approach: using advanced slicing methods

```
with open('input.txt') as f:
  letters = [line[1::4] for line in f]
```

5. Use a class attribute to store class instances

On Day11 a situation is described in which monkies pass objects to each other. In order to simplify we'll pretend that they're simply passing bananas to each other. Each monkey can be represented as an instance of a Python `class` with their `id` and their amount of bananas as instance attributes. However, there are many monkeys and they need to be able to interact with each other. A trick to store all the monkeys and for them to be able to interact with each other is to define a dictionary with all `Monkey` instances as a class attribute of the `Monkey` class. Using `Monkey.monkeys[id]` you can access all existing monkies without the need of a `Monkies` class or an external dictionary:

```
class Monkey:
  monkeys: dict = dict()

  def __init__(self, id: int):
      self.id = id
      self.bananas = 3
      Monkey.monkeys[id] = self

  def pass_banana(self, to_id: int):
      Monkey.monkeys[to_id].bananas += 1
      self.bananas -= 1

Monkey(1)
Monkey(2)
Monkey.monkeys[1].pass_banana(to_id=2)

print(Monkey.monkeys[1].bananas)
2

print(Monkey.monkeys[2].bananas)
4
```

6. Self-documenting expressions (BONUS)

This trick is applicable virtually every time you write a Python program. Instead of defining in an f-string what you are printing (e.g. `print(f"x = {x}")` you can use `print(f"{x = }")` to print the value with a specification of what you are printing.

```
# INPUT
x = 10 * 2
y = 3 * 7

max(x,y)

# DESIRED OUTPUT
x = 20
y = 21

max(x,y) = 21
```

Junior developer approach:

```
print(f"x = {x}")
print(f"y = {y}")

print(f"max(x,y) = {max(x,y)}")
```

Senior developer approach:

```
print(f"{x = }")
print(f"{y = }")

print(f"{max(x,y) = }")
```

To conclude

We've looked at 5 Python tricks that distinguish senior developers from junior developers. Of course, only applying these tricks won't promote someone suddenly to senior developer. However, through analyzing the difference in style and pattern between the two you can learn the difference in how a senior developer approaches coding problems versus a junior, and you can start to internalize these approaches so that you'll eventually become a senior developer yourself!

. . .

If you liked this article and want to learn more about senior Python approaches, make sure to read e.g. my other article about how to extract more information from categorical plots:

How to extract more information from categorical plots

Easily get deeper insights into your categorical data by using these two methods.

towardsdatascience.com

• • •

Resources

- List comprehensions
- Map explanation
- Enumerate documentation
- AoC Reddit (hints, solutions, and discussions)

Python Programming Python Coding Data Science Programming