Today: unify lines with variable, boolean precedence, string functions, unicode

Today - Protip - Better / Shorter

Today, I'll show you a protip, starting with correct code, re-writing it in a better, shorter way. It's intuitively satisfying to have a 10 line thing, and shrink it down to 6 lines that reads better.

First: Variable Set Default Pattern

This is a handy pattern to set a variable according to a boolean. (1) Initialize (set) the variable to its default, common value first. (2) Then an if-statement detects if we need to initialize it to a different value.

```
alarm = '8:00 am'
if is_weekend:
    alarm = ''
# Alarm is now set, one way or another.
# Lines below just use it.
```

Technique: Better/Shorter - Unify Lines

> Better problems

Unify Lines with a Variable

```
if case-1:
    lines-a
    ...
    ...

if case-2:
    lines-b
```

```
        ...
        ...
```

- Suppose we have
  case-1 .. solved by lines-A
  case-2 .. solved by lines-B
- But lines-A and lines-B are similar
- Maybe did copy/paste lines-A and lines-B
- **Unify** - re-structure the code so one set of lines works for both cases
- Move the difference between lines-A and lines-B into a variable
- Hard to describe in the abstract, let's look at an example

Aside: Copy / Paste

- Copy/paste of some lines can be ok
- Sometimes lines of code are very similar
- e.g. Doctests
- Common bug: paste lines in, but forget to update in some spot
- **But**: with copy/paste .. we wonder if there's a way to unify, not have two sets of lines

speeding() Example

> speeding()

speeding(speed, is_birthday): Compute speeding ticket fine as function of speed and is_birthday boolean. Rule: speed under 50, fine is 100, otherwise 200. If it's your birthday, the allowed speed is 5 mph more. Challenge: change this code to be shorter, not have so many distinct paths.

The code below works correctly. You can see there is one set of lines each for the birthday/not-birthday cases. What exactly is the difference between these two sets of lines?

```
def speeding(speed, is_birthday):
    if not is_birthday:
        if speed < 50:
            return 100
        return 200

    # is birthday
    if speed < 55:
        return 100
    return 200
```

Unify Cases Solution

- The 2 "speed" if-statements look really similar
- They differ by the value in the if-test **50** vs. **55**
- Solution: introduce variable: `limit`
- 1. If/logic sets limit for the various cases
- 2. Unified code below just uses limit, works for all cases

speeding() Better Unified Solution

1. Set limit first. 2. Then unified lines below use limit, work for all cases.

```
def speeding(speed, is_birthday):
    limit = 50
    if is_birthday:
        limit = 55
    if speed < limit:
        return 100
    return 200
```

Example ncopies()

> ncopies()

```
ncopies: word='bleh' n=4 suffix='@@' ->

    'bleh@@bleh@@bleh@@bleh@@'
```

Change this code to be better / shorter. Look at lines that are similar - make a unified version of those lines.

ncopies(word, n, suffix): Given name string, int n, suffix string, return n copies of string + suffix. If suffix is the empty string, use `'!'` as the suffix. Challenge: change this code to be shorter, not have so many distinct paths.

Before:

```
def ncopies(word, n, suffix):
    result = ''

    if suffix == '':
        for i in range(n):
            result += word + '!'
    else:
        for i in range(n):
            result += word + suffix
    return result
```

ncopies() Unified Solution

Solution: use logic to set an `ending` variable to hold what goes on the end for all cases. Later, unified code uses that variable vs. separate if-stmt for each case. Alternately, could use the `suffix` parameter as the variable, changing it to `'!'` if it's the empty string.

```
def ncopies(word, n, suffix):
    result = ''
    ending = suffix
    if ending == '':
        ending = '!'

    for i in range(n):
        result += word + ending
    return result
```

(optional) match()

> match()

match(a, b): Given two strings a and b. Compare the chars of the strings at index 0, index 1 and so on. Return a string of all the chars where the strings have the same char at the same position. So for 'abcd' and 'adddd' return 'ad'. The strings may be of any length. Use a for/i/range loop. The starter code works correctly. Re-write the code to be shorter.

Before:

```
def match(a, b):
    result = ''
    if len(a) < len(b):
        for i in range(len(a)):
            if a[i] == b[i]:
                result += a[i]
    else:
        for i in range(len(b)):
            if a[i] == b[i]:
                result += a[i]
    return result
```

match() Unified Solution

```
def match(a, b):
    result = ''
    # Set length to whichever is shorter
    length = len(a)
    if len(b) < len(a):
        length = len(b)

    for i in range(length):
        if a[i] == b[i]:
            result += a[i]

    return result
```

Reminder Avoid: `== True`, `== False`

```
# have a "is_raining" boolean var


if is_raining == True:  # NO do not write
this
    ...


if is_raining:          # YES, this way
   ...
```

Boolean Expressions

See the guide for details Boolean Expression

- Boolean operators: `and or not`
- Mixture of these, can add parenthesis to force order of operation "precedence" in CS parlance
- Say have three boolean variables, each True/False
  `age` - say age is good if less than 30
  `is_raining` - True if raining
  `is_weekend`- True if it's the weekend
- Define: to be a good day, need two things:
  1. it must not be raining
  2. then either age is under 30 or it's the weekend

The code below looks reasonable, but doesn't quite work right

```
def good_day(age, is_weekend, is_raining):
    if not is_raining and age < 30 or
is_weekend:
        print('good day')
```

Boolean Precedence:

- `not` = highest, (like - in -7)
- `and` = next highest (like *)
- `or` = lowest (like +)

What The Above Does

Because **and** is higher precedence than **or** as written above, the code above acts like the following (the and going before the or):

```
    if (not is_raining and age < 30) or
is_weekend:
```

You can tell the above does not work right, because any time `is_weekend` is True, the whole thing is True, regardless of age or rain. This does not match the good-day definition above, which requires that it not be raining.

Boolean Precedence Solution

The solution we will spell out is not difficult.

- Many programmers do not have boolean precedence memorized .. fine
- Do remember that "not" is the highest precedence
- Solution: note when you have a mixture of **and** + **or**
  When there is a mixture, the precedence will matter
  put in parenthesis in that case
- We will never complain about extra parenthesis, so just add them to spell out the order you want
- In this case, put parens to group the **or** part, separating from not-raining
- BTW similar logic applies to math - if there's a mixture of * and +, add parenthesis

Solution

```
def good_day(age, is_weekend, is_raining):
    if not is_raining and (age < 30 or
is_weekend):
        print('good day')
```

String - More Functions

See guide for details: Strings

Thus far we have done String 1.0: len, index numbers, upper, lower, isalpha, isdigit, slices, .find().

There are more functions. You should at least have an idea that these exist, so you can look them up if needed. The important strategy is: don't write code manually to do something a built-in function in Python will do for you. The most important functions you should have memorized, and the more rare ones you can look up.

s.startswith() s.endswith()

These are very convenient True/False tests for the specific case of checking if a substring appears at the start or end of a string. Also a pretty nice example of function naming.

```
>>> 'Python'.startswith('Py')
True
>>> 'Python'.startswith('Px')
False
>>> filename = 'resume.docx'
>>> filename.endswith('.docx')
True
```

String - replace()

- `str.replace(old, new)`
- Returns a new string with replacements done (immutable)
- Does not respect word boundaries, just dumb replacement
- Aside: Anti-Pattern
  Trying to compute something about s
  e.g. count the digits in s

Do not use replace() to modify s as a shortcut to computing about s
Not a good strategy

```
>>> s ='this is it'
>>> s.replace('it', 'odd')
'this is odd'
>>>
>>> s.replace('is', 'xxx')  # replaces
anywhere
'thxxx xxx it'
>>>
>>> s.replace('is', '')  # replace w/ empty
str
'th  it'
>>>
>>> s        # s not changed
'this is it'
```

Recall: `x = change(x)`

Recall how calling a string function does not change it. We had the `x = change(x)` pattern, computing a new string and storing it back into the string variable.

```
# NO: Call without using result:
s.replace('is', 'xxx')
# s is the same as it was


# YES: this works
s = s.replace('is', 'xxx')
```

String - strip()

- Removes whitespace chars from either end
- Use inside `for line in f` to trim off `'\n'

```
>>> s = '    this and that\n'
>>> s.strip()
'this and that'
```

String - split()

- Nice feature to parse a line of text
  e.g. from a file line `11,45,19.2,N`
- `str.split()` -> array of strings
- `str.split(',')` - split on `','` substring
- `str.split()` - with zero parameters
  a special form of split()
  splits on 1 or more whitespace chars
  combines multiple whitespace chars
  handy primitive "word" from line feature

```
>>> s = '11,45,19.2,N'
>>> s.split(',')
['11', '45', '19.2', 'N']
>>> 'apple:banana:donut'.split(':')
['apple', 'banana', 'donut']
>>>
>>> 'this    is     it\n'.split()   #
special whitespace form
['this', 'is', 'it']
```

String - join()

- Reverse of split()
- Given list of strings, puts them together to make a big string
- Mnemonic: str.split() and str.join()
  The string is the noun in noun.verb form

```
>>> foods = ['apple', 'banana', 'donut']
>>> ':'.join(foods)
'apple:banana:donut'
```

Basic String Assembly: + and str()

- You have some string data and some other data
- Want to put them together to form a string
  Making a string, say, to put on screen
- Basic technique: use +, call str() function
- This is a fine way to do it (new way below)

```
>>> name = 'Alice'
>>> score = 12
>>> name + ' got score:' + str(score)
'Alice got score:12'
>>>
```

Format String - New

Put a lowercase 'f' to the left of the string literal, making a specially treated "format" string. For each `{..}` in the string, Python evaluates the expression within and pastes the resulting value into the string there. Super handy! The expression has access to local variables.

```
>>> name = 'Alice'
>>>
>>> f'this is {name}'
'this is Alice'
>>>
>>> score = 12
>>> f'{name} got score:{score}'
Alice got score:12
>>>
```

Optional: Limit Digits `{x:.4}`

Add `:.4` after the value in the curly braces to limit decimal digits printed. There are many other format options, but this is the one I use the most by far.

```
>>> x = 2/3
>>> f'value: {x}'
'value: 0.6666666666666666'
>>> f'value: {x:.4}'   # :.4
'value: 0.6667'
```

String Unicode

In the early days of computers, the ASCII character encoding was very common, encoding the roman a-z alphabet. ASCII is simple, and requires just 1 byte to store 1 character, but it has no ability to represent characters of other languages.

Each character in a Python string is a unicode character, so characters for all languages are supported. Also, many emoji have been added to unicode as a sort of character.

Every unicode character is defined by a unicode "code point" which is basically a big int value that uniquely identifies that character. Unicode characters can be written using the "hex" version of their code point, e.g. "03A3" is the "Sigma" char Σ, and "2665" is the heart emoji char ♥.

Hexadecimal aside: hexadecimal is a way of writing an int in base-16 using the digits 0-9 plus the letters A-F, like this: **7F9A** or **7f9a**. Two hex digits together like 9A or FF represent the value stored in one byte, so hex is a traditional easy way to write out the value of a byte. When you look up an emoji on the web, typically you will see the code point written out in hex, like 1F644, the eye-roll emoji 🙄.

You can write a unicode char out in a Python string with a `\u` followed by the 4 hex digits of its code point. Notice how each unicode char is just one more character in the string:

```
>>> s = 'hi \u03A3'
>>> s
'hi Σ'
>>> len(s)
4
>>> s[0]
'h'
>>> s[3]
'Σ'
>>>
>>> s = '\u03A9'   # upper case omega
>>> s
'Ω'
>>> s.lower()      # compute lowercase
'ω'
>>> s.isalpha()    # isalpha() knows about
unicode
True
>>>
```

```
>>> 'I \u2665'
'I ♥'
```

For a code point with more than 4-hex-digits, use `\U` (uppercase U) followed by 8 digits with leading 0's as needed, like the fire emoji 1F525, and the inevitable 1F4A9.

```
>>> 'the place is on \U0001F525'
'the place is on 🔥'
>>> s = 'oh \U0001F4A9'
>>> len(s)
4
```

Ethic: Have Some Generosity In Your Life

- Your life goal is not to consume everything just for yourself
- Part of your life is contributing to others
  Most closely, your family
  But the circle of people to contribute to extends outwards
  Ultimately including people around the world
- You do not need to live a vow of poverty
  Plenty of middle ground here
  Have some generosity, including to people you don't know
- Aside: happiness research:
  Little generous acts, a source of personal happiness

History of Unicode and Python

The history of ASCII and Unicode is an example of generosity ethics.

ASCII

In the early days of computing in the US, computers were designed with the ASCII character set, supporting only the roman a-z alphabet. This hurt the rest of the planet, which mostly doesn't write in English. There is a well known pattern where technology comes first in the developed world, is scaled up and becomes inexpensive, and then proliferates to the developing world. Computers in the US using ASCII hurt that technology pipeline. Choosing a US-only solution was the cheapest choice for the US in the moment, but made the technology hard to access for most of the world. This choice was practical with the costs of the time, but also ungenerous.

Unicode Technology

Unicode takes 2-4 bytes per char, so it is more costly than ASCII. Cost per byte aside, Unicode is a good solution - a freely available standard. If a system uses Unicode, it and its data can interoperate with the other Unicode compliant systems.

Unicode vs. RAM Costs vs. Moore's Law

The cost of supporting non-ASCII data can be related to the cost of the RAM to store the unicode characters. In the 1950's every byte was literally expensive. An IBM model 360 could be leased for $5,000 per month, non inflation adjusted, and had about 32 kilobytes of RAM (not megabytes or gigabytes .. kilobytes!). So doing very approximate math, figuring RAM is half the cost of the computer — $2500 —, we get a cost of about $1 per byte per year.

```
>>> 2500 * 12 / 32000
0.9375
```

So in 1950, Unicode is a non-starter. RAM is too expensive.

Then we have Moore's law - chip capacity doubling every 2 years. An exponential increase in what a chip can store.

RAM Costs Today

What does the RAM in your phone cost today? Just for round numbers, let's say the RAM cost of your phone is $500 and your phone has 8GB of RAM (conservative). What is the cost per byte?

The figure 8 GB is 8 billion bytes. In Python, you can write that as `8e9` - like on your scientific calculator.

```
>>> 500 / 8e9   # 8 GB
6.25e-08
>>>
>>> 500 / 8e9 * 100  # in pennies
6.2499999999999995e-06
```

RAM costs nothing today - 6 millionths of a cent per byte. This is the result of Moore's law. Exponential growth is incredible.

Unicode Makes Sense in 1990s

Sometime in the 1990s, RAM was cheap enough that spending 2-4 bytes per char was not so bad, and around then is when Unicode was created. Unicode is a standard way of encoding chars in bytes, so that all the Unicode systems can transparently exchange data with each other.

With Unicode, the tech leaders were showing a little generosity to all the non-ASCII computer users out there in the world.

Generosity and Python Story

- Python created by Guido van Rossum
- From the Netherlands
- Where they speak Dutch
- What language encoding was used for Python?
- Unicode, of course

- Therefore, Python works with data in Palo Alto, Tokyo, Stockholm .. everywhere

With Unicode, there is just one Python that works in every country. A world of programmers contribute to Python as free, open source software. We **all** benefit from that community, vs. each country maintaining their own in-country programming language, which would be a crazy waste of duplicated effort.

Ethic: Generosity

So being generous is the right thing to do. But the story also shows, that when you are generous to the world, that generosity may well come around and help yourself in the end.