17+ Advanced Python Hacks To Help You Pass Interviews And Be More Confident In Your Python Skills NOW

Powerful Python

https://powerfulpython.com

1. Class Methods

Use `@classmmethod` to create methods on classes:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

    @classmethod
    def from_pennies(cls, pennies):
        dollars = pennies // 100
        cents = pennies % 100
        return cls(dollars, cents)
```

Benefit: Write handy "alternate constructors" to instantiate your class from special data representations, or handle extra one-time configuration. And it automatically extends to subclasses!

2. Avoid This Diabolical Antipattern

When you catch exceptions, never do this:

```
# NO!!!
try:
   # ...
except:
   pass
```

Writing "except: pass" creates horrible bugs that are almost impossible to figure out.

Instead, always make your except block catch the most specific exception available that will do what you need. And NOT catch anything else.

Look carefully at this code - can you spot the hidden bug:

```
fruit_info = {"name": "strawberries", "calories": 33, "protein": 0.7}

try:
    print(fruitinfo["carbohydrates"])
except KeyError:
    print("carbohydrate info not available")
```

The dictionary is `fruit_info`, but inside the try block, we forget the underscore. This creates a `NameError`, which is NOT caught by the except block. That's great! We can see the bug right away and fix it immediately.

If we wrote "except: pass", we might not notice this bug until we already shipped the code!

3. Make Functions More Efficient With `yield`

This function returns a list - a large one, if `limit` is large:

```
def make_squares(limit):
    squares = []
    for num in range(limit):
        squares.append(num**2)
    return squares
```

Instead, use `yield`:

```
def gen_squares(limit):
    for num in range(limit):
        yield num**2
```

Benefit: This is more memory efficient and potentially more responsive, as it gives you one element at a time to process, instead of insisting on creating ALL of them in a list all at once.

4. Use Python's Built-In Exceptions

The two most important built-in exceptions to understand: `TypeError` and `ValueError`.

`TypeError` means you passed in the wrong type of data:

```
>>> # int() can take a number or a string, but not a list
>>> int([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-like object or a number, not
'list'
```

`ValueError` means you passed in an acceptable type, but this particular value is not acceptable:

```
>>> # If you give int() a string, it must represent a number.
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Benefit: More quickly understand the error messages you see, and how to quickly fix them.

Bonus: When raising exceptions in your own functions and methods, using an exception the way Python does makes your code easier to work with and understand.

5. List Comprehensions

Instead of this:

```
>>> squares = []
>>> for n in range(5):
...     squares.append(n**2)
...
>>> print(squares)
[0, 1, 4, 9, 16]
```

Write this:

```
>>> squares = [ n**2 for n in range(5) ]
>>> print(squares)
[0, 1, 4, 9, 16]
```

Benefit: More readable and expressive data structures, in a way that is easy to understand even for developers who have never seen list comprehensions before.

6. Key Functions

Pass a "key" function to `sorted()`, `max()` and `min()`, to alter their default comparison mode:

```
>>> # A list of strings, that we want to sort by their numerical value.
>>> nums = ["7", "12", "9", "3"]
>>>
>>> # By itself, sorted() does not sort them like we want.
>>> sorted(nums)
['12', '3', '7', '9']
```

```
>>> # Pass a "key function" to tell it now to sort correctly.
>>> sorted(nums, key=int)
['3', '7', '9', '12']
>>>
>>> # Also works with max() and min().
>>> max(nums, key=int)
'12'
>>> min(nums, key=int)
'3'
```

Benefit: Clearer and shorter code that uses Python's built-in tools in powerful and useful ways.

7. Unit Tests

Write automated tests to verify your code stays correct:

```
from unittest import TestCase

def to_url(phrase):
    return "/" + phrase.lower().replace(" ", "-") + ".html"

class TestForFunction(TestCase):
    def test_to_url(self):
        self.assertEqual(
            "/how-to-level-up-your-python.html",
            to_url("How To Level Up Your Python"))
```

Benefit: Higher quality, more robust code. And makes it possible for you to create powerfully complex software that was just too overwhelming before.

8. F-strings

F-strings let you create strings from local variables:

```
>>> name = "John"
>>> dollars = 3
>>> cents = 5
>>> food = "DONUTS"
>>>
>>> f"{name} gave me ${dollars}.{cents:02} for {food.lower()}"
'John gave me $3.05 for donuts'
```

Benefit: Render strings intuitively and concisely. Also clearly demonstrates you are up-to-date on Python, as f-strings are a relatively recent addition to the language.

9. Properties

Create methods that act like member variables with @property:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

    @property
    def total_cents(self):
        return 100 * self.dollars + self.cents
```

```
>>> cash = Money(42, 17)
>>> cash.total_cents
4217
```

(Notice there is no "(" after `cash.total_cents`. It’s acting like a member variable, even though you defined it as a method.)

Benefit: Dynamically calculated values, data validation, denormalizing, and can sometimes also help in refactoring.

10. "Finally" Blocks

When you have `try` block, use a `finally` block to make sure something happens no matter what:

```
job = MachineLearningJob(job_spec)
cloud_vm = VirtualMachine()

# Start the virtual machine
cloud_vm.start()
try:
    # train our ML model
    cloud_vm.run_job(job)
finally:
    # Terminate the VM so it is not using resources
    cloud_vm.stop()
```

Even if `run_job()` raises a program-crashing exception, the `stop()` method still gets called.

11. Lambdas

Imagine you need to find the highest-protein food in this data set:

```
# Per 100 grams.
foods = [
    {"name": "strawberries", "calories": 33, "protein": 0.7},
    {"name": "waffles", "calories": 291, "protein": 8},
    {"name": "eggs", "calories": 155, "protein": 13},
    {"name": "chicken breast", "calories": 165, "protein": 31},
]
```

You can of course define a key function, and use it with `max()`:

```
>>> def protein_of(item):
...     return item["protein"]
>>> max(foods, key=protein_of)
{'name': 'chicken breast', 'calories': 165, 'protein': 31}
```

Even better, use a "lambda" - an anonymous, in-line function:

```
>>> max(foods, key=lambda item: item["protein"])
{'name': 'chicken breast', 'calories': 165, 'protein': 31}
```

Benefit: No need to define a function you will only use once. Different readability trade-offs, which - when used wisely - creates more readably concise code.

12. Parse Dates With python-dateutil

Dates are horrible to parse with Python’s built-in `datetime` module. So many wildly different formats:

```
timestamps = [
    "02/05/30 11:19pm EST",
    "6/8/31 22:35",
    "Mon, Sep 16, 2019, 11:52 AM",
    "6/24/2031  7:13:44 AM",
    "5/15/2030",
    "2031-05-12 13:45:17",
]
```

Solution: install the `dateutil` module, and use its `parse()` function:

```
>>> from dateutil.parser import parse
>>> for timestamp in timestamps:
...     when = parse(timestamp)
...     # "when" is now a standard instance of Python's built-in datetime type.
...     print(timestamp + "\n  -> " + repr(when))

02/05/30 11:19pm EST
  -> datetime.datetime(2030, 2, 5, 23, 19)
6/8/31 22:35
  -> datetime.datetime(2031, 6, 8, 22, 35)
Mon, Sep 16, 2019, 11:52 AM
  -> datetime.datetime(2019, 9, 16, 11, 52)
6/24/2031  7:13:44 AM
  -> datetime.datetime(2031, 6, 24, 7, 13, 44)
5/15/2030
  -> datetime.datetime(2030, 5, 15, 0, 0)
2031-05-12 13:45:17
  -> datetime.datetime(2031, 5, 12, 13, 45, 17)
```

Benefit: Easily handles any date and time format you will ever encounter - automatically, easily and accurately.

13. Use `*args`

Write functions that can handle any number of arguments with `*args`:

```
def read_text_files(*paths):
    text = ""
    for path in paths:
        with open(path) as text_file:
            text += text_file.read()
    return text

# Reads in the content of all three .txt files,
# and concatenates them together.
full_text = read_text_files("a.txt", "b.txt", "c.txt")
# Can take any number of arguments.
more_text = read_text_files("z.txt")
even_more_text = read_text_files("this.txt", "that.txt")
```

Benefit: Flexible function and method interfaces that allow you to write more powerfully generic code.

14. Data Classes

Here's the `Money` class again:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

    @classmethod
    def from_pennies(cls, pennies):
        dollars = pennies // 100
        cents = pennies % 100
        return cls(dollars, cents)
```

A better way to write it is using the `@dataclass` decorator:

```
from dataclasses import dataclass

@dataclass
class Money:
    dollars: int
    cents: int

    @classmethod
    def from_pennies(cls, pennies):
        dollars = pennies // 100
        cents = pennies % 100
        return cls(dollars, cents)
```

Both versions work exactly the same:

```
>>> cash = Money.from_pennies(7249)
>>> print(cash.dollars)
72
>>> print(cash.cents)
49
```

Benefit: Save time and reduce error with repetitive `__init__` definitions, plus you are documenting the intended types.

15. Rename Imports

When you import modules, classes, functions or objects, you can rename them with an "as" clause:

```
# Lets you call the parse() function under the name parse_datetime()
from dateutil.parser import parse as parse_datetime
```

Benefit: More readable and intuitive names. In this example, the name `parse()` is confusingly ambiguous. Parse what? JSON? XML? Python source code? But `parse_datetime()` is clear and takes no mental energy to remember what it is for.

16. Logging Errors

Use logging in your code to get more insight into what your application does during runtime:

```
import logging

def get_customer(customer_id, customer_list):
    try:
        customer = customer_list[customer_id]
    except IndexError:
        logging.error("Cannot find customer with ID %s", customer_id)
        customer = None
    return customer
```

Note that even in modern Python, you must use "percent formatting" with logging, and not f-strings or str.format().

Benefit: Useful information about what the program is doing at runtime, even in production - for telemetry and monitoring, and also to quickly troubleshoot otherwise horrendous bugs.

17. Use Magic Methods

See this class:

```
from datetime import date

class DateRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        assert self.start <= self.end

    def in_range(self, when):
        return self.start <= when <= self.end
```

You can use it like:

```
# Valentine's day in 2104!
vday = date(2104, 2, 14)
# Party like it's 1999
eve = date(1999, 12, 31)

>>> dr.in_range(vday)
True
>>> dr.in_range(eve)
False
```

`in_range()` is handy, but it would be nice if we could use Python’s built-in "in" operator:

```
>>> nums = [1, 2, 3]
>>> 2 in nums
True
>>> 10 in nums
False
```

Good news, you can! Just add a method called `__contains___()`:

```
class DateRange:
    # ...

    def __contains__(self, when):
        return self.start <= when <= self.end
```

Now it works:

```
>>> vday in dr
True
>>> eve in dr
False
```

How cool is that!

Python has many "magic methods" like this, all starting and ending with two underscores. Each lets you "hook" into Python’s syntax in some fun, useful way like this.

18. How To Get Help

If you want to:

- Get more interviews for great Python jobs,
- Pass those interviews more easily,
- Learn the high-level Python skills that earn you a higher salary,
- Not to mention the joy of creating more powerful and interesting software…

If that is you, Powerful Python can help. Learn more here:

https://powerfulpython.com

Powerful Python