POWERFUL PYTHON PLAYBOOK

How To Finally Blast "Beyond The Basics" And Master The Most Important Programming Language On Earth

Hi, I'm Aaron Maxwell – author of the book "Powerful Python: The Most Impactful, Patterns, Features and Development Strategies Modern Python Provides."

I made this Playbook to help EVERYONE who's ready to go "beyond the basics" of Python…

Because there are lots of intro courses and tutorials out there… But **YOU are already past that "beginner" Python level…**

In fact, if you are *highly motivated* and ready to:

- FINALLY master Python at that deep, masterful, "Pythonista" level…
- Learn the skills to increase your annual earnings by $10k… $20k… $50k or more, for the rest of your career…
- Get on the road to being one of the most highly regarded developers on your team…
- And do it all in 6 weeks or less…

… then book a time to talk to us here.

Read Part 1 of this guide, to get a *high level strategy* for mastering Python quickly… and Part 2 where we look at more advanced Python code, with specific coding exercises for you…

Part 1: Strategy For Python Mastery

Before we “dive in” to some intermediate Python code, and give you some mind-stretching coding exercises…

Let’s step back. And look at a high-level strategy for continually improving your skill with Python… to the intermediate and advanced levels, and BEYOND.

Starting with what NOT to do… then the 4 pillars of advanced Python mastery….

How NOT To Do It #1: Mass-Market Courses

Ever looked excitedly at the description of a Python course, skimming its list of topics….

Only to feel disappointment… when you realize it doesn't cover ANYTHING you don't already know?

I'm sure you've noticed this. For Python, courses and books rarely go much further than the beginner-basic level… **Which means they can't help you.**

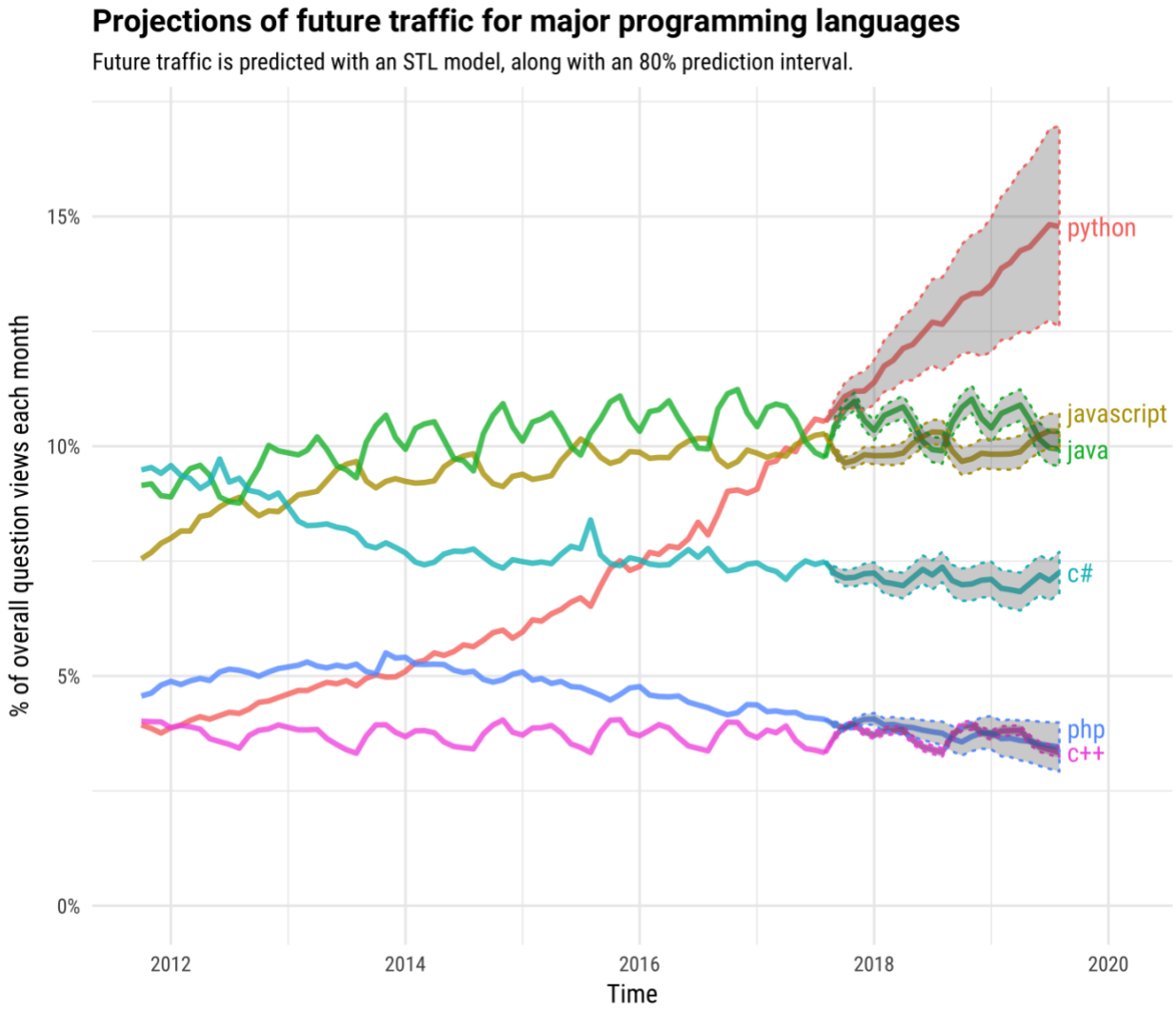
Even worse, when you DO finally find a course teaching you something new…

Have you found the course instructor just isn't very good?

It's a horrible feeling to listen to an explanation four times, feeling stupid because you just don't get it…

But it's not your fault. Coding is a skill, and teaching is a skill. Not everyone who can code also knows how to teach.

Here's a chart illustrating Python's exploding growth, and how it's on track to become THE dominant programming language:



(Source: Stack Overflow Blog, "The Incredible Growth of Python")

We're at a time where true mastery of this language can open incredible doors for you... so it's important to not miss this once-in-a-lifetime opportunity.

To ensure you finally go far beyond the basics of Python, in 6 weeks or less, book a time to talk to us here.

How NOT To Do It #2:
Relying On Online Posts

Ever felt confused about how to do something in your code?

So you search, and find a promising post on Stack Overflow, a relevant discussion on Reddit, or even a blog post some developer wrote up...

And it helps. Especially for "let's just get this working so we can move on" kind of information...

But if you're reading this, you're not the kind of developer who wants to just copy-paste something, without understanding how it works.

You want a deeper understanding and mastery of Python...

That's why it's so frustrating how many of these posts have serious shortcomings:

- They're "shaded" and biased by the writer's opinion or agenda.
- They're dated or partly obsolete.
- The post has a working solution... but it's "brittle". It'll break in situations not exactly like the poster is describing... and yours isn't exactly the same.

Again, posts like these are useful, and great to have.

But if you want wide and deep mastery of Python, without wasting years hitting dead ends...

You need something more.

To ensure you master Python fast, book a time to talk to us here.

How NOT To Do It #3:
"Fake It Till You Make It"

Have you ever written some code, and it works... More or less...

But you feel like you don't understand WHY?

There's nothing wrong with doing this once in a while. We all have too much to learn, with not enough time. So we have to pick our battles.

Sometimes we just write some code we don't completely grok – maybe even copy-pasting it...

Tweak it a bit, and move on to the next problem we are already behind schedule solving.

But:

If you're having this don't-get-it feeling a lot... That's a sign to deepen your understanding.

When you deeply understand the fundamental principles of Python, for example, then YOU become the one who can easily write answers to questions people ask.

And of course, this lets you write code better and faster yourself.

The New And Proven Way To Catapult Your Python Skills To The Advanced Levels FAST, Without Wasting Time On What Doesn't Matter

Have you ever devoted days, weeks, maybe even months of self-study to learn a new programming topic...

Only to struggle every step of the way, and end up feeling like you didn't understand even half of what you wanted to learn?

It's heartbreaking.

Over the past few years, I've taught thousands of working developers around the globe how to break through to the intermediate and advanced levels of Python programming...

And through that experience, I've found that heartbreaking struggle often has a root cause.

Simply put, there are certain fundamental steps that haven't been followed. Steps that, once you walk through them, seemingly "open up" your skills with Python to a whole new level.

So you can learn new technologies based on Python faster and easier. And of course, write better code faster, more complex applications more easily, than ever before.

To ensure you master Python fast, book a time to talk to us here.

Pillar 1. Complex Software Toolkit

These are the "key skills" for successfully writing complex applications in every language...

And how they work in Python, specifically.

I'm talking about the skills, techniques, thinking strategies, methodologies that apply all great software development. And I emphasize this for two reasons.

First, many people using Python today did not come through a traditional software engineering background.

And naturally, as a result, they never picked up those traditional skills.

Maybe you're in that group, maybe not. Because the SECOND reason I bring this up...

Is that you also need to express these ideas "Pythonically". To get the most out of this high- powered language.

So doing all that - that's Step 1.

Pillar 2. The "First Principles of Python"

Have you heard of the idea of "first principles"?

This is the idea behind many of the great scientific advances in human history. It goes all the way back to at least Aristotle...

Was used every day by 20th-century scientific giants like Richard Feynman...

And today, titans like Elon Musk swear by it as a key to their massive, mind-blowing success.

When you learn to think from "first principles", the benefit is clear:

It unleashes your creativity.

That's why I emphasize what I call the "First Principles of Python". Because once you understand them well...

You can - almost without effort - combine them and build on them, to solve complex, hair-raising problems that will....

Send other developers running away screaming!!!

Bonus: it's also a LOT of fun!

And it leads into the next step...

Pillar 3. Seeing Into The “Matrix” of Python

You've seen the movie "The Matrix", right?

Even if you haven't, you've probably seen a clip from the movie, in a hallway with nowhere to run... these bad dudes in suits and shades fire their pistols at the trench-coat-wearing hero...

And he simply puts up his hands, and stops those bullets in mid-flight.

He's not dodging the bullets. He doesn't HAVE to...

Because he can see into the very fabric of reality, feel into the structure of our universe, and manipulate it at his will.

Another way to say this: he can do what others cannot do... because he sees what others cannot see. And knows things others do not know.

Something similar can happen with your Python skill.

Where your mastery of the fundamentals of software development, combine with your mastery of the First Principles of Python...

You can look at any Python application, any bit of Python code... And see things other developers don't see.

You can think about a problem that needs to be solved by writing code, and easily imagine how to gracefully, easily, powerfully express that solution in Python.

Stopping metaphorical bullets in their tracks. It's great.

(Remember the last time you spent all day figuring out a stupid, frustrating bug? That's the kind of thing "seeing into the Matrix of Python" lets you AVOID. How does THAT sound to you?)

Pillar 4. The "Amplify Effect"

How can you have the most impact?

How can you maximize the positive impact of the code you write. So

what you're doing is not only fun, and satisfying... Not only pays the bills...

But makes a difference in the world... in a way you can clearly see?

Now, I don't mean creating a famous open- source project that everyone and their brother starts using. Though that's possible, if you're dedicated enough.

What's more important, in my view:

That you're the kind of programmer who elevates the success of your team... Whatever group of programmers you find yourself in, whatever organization or company you're involved with...

And part of that is organizing the code you write...

So that you're solving the hardest, most burdensome problems your team or company faces... And solving it well...

AND:

Doing it in a way that's easy for other developers to reuse. To leverage, integrate and use in their own code. There are several parts to this, some of which you may not expect.

And the effect of all this is to AMPLIFY the positive impact of the code you write.

This benefits everyone who works with you... Making you a positive force on the team...

And more your-self-servingly, it also enhances your reputation, your career...

Ultimately, in all likelihood, leading to you making more MONEY.

There are 4 steps we use in our system to create a predictable, consistent, and rapid improvement process for our students.

My hope is that this playbook will give you clarity, and give you a better idea of what REALLY works... breaking you through to that level of Python mastery you've always wanted, but has been frustratingly just out of reach.

This system eliminates all of the guesswork in getting in that top 1% of "elite" Python developers, no matter what you're using Python for... and stay solidly ahead of the curve for years to come.

If you'd like to speak with us on how we can help you use this powerful system, we'd love to talk to you.

There is never any pressure on our clarity sessions. Our objective is to figure out where you're having the most trouble breaking through as a Python coder, and then help you figure out what the best plan of action is...

And if you'd like to set up a clarity session with us, follow the URL below to do that:

powerfulpython.com/apply

Feel free to send us an email (if you need anything) or if you just want to say 'thanks for the Playbook'.
(service@powerfulpython.com)

Looking forward to talking soon!

Aaron Maxwell
Founder, Powerful Python

Part 2:
Three Keys to Level Up Your Python

Key #1: Scalability

As someone who's already mastered Python's basics, what do you focus on next? In this guide, you'll learn which aspects of Python catapult you to your next level of mastery - and get more out of the language than you ever thought possible.

The first key: **scalability**. We're now in the age of Big Data. And it's not just a buzzword; from now on, all of us - no matter what we're working on - need to be able to write software that can handle increasing magnitudes of data.

In other words, the code you write needs to be *scalable*. Removing hidden memory bottlenecks, so performance doesn't grind to a halt - or worse, your program is brutally killed by the operating system, because it's using too much memory.

The most scalable systems are in the realm of distributed computing. But the idea applies to programs that run on a single computer, and for almost all of us, that's more important in our day to day work. So let's focus on Python's tools for it.

Exercise: Bottlenecks To Scalability

Here's some Python code:

```
def fetch_squares(max_root):
    squares = []
    for x in range(max_root):
        squares.append(x**2)
    return squares

MAX = 5
```

```
for square in fetch_squares(MAX):
    do_something_with(square)
```

1) Do you see a memory bottleneck here? If so, what is it?

2) Can you think of a way to fix the memory bottleneck?

Pause here to really think about the code, and these questions, before you continue.

Making Your Python Code Scalable

Let's look at the “fetch_squares” function again:

```
def fetch_squares(max_root):
    squares = []
    for x in range(max_root):
        squares.append(x**2)
    return squares
```

While it works, here's the big problem. What if MAX is not five, but five million? Or five billion, or trillion, or more?

That's right: your memory usage goes through the roof, because “fetch_squares()” creates an entire list. That's a tragedy, because the individual square numbers are only needed one at a time. As MAX increases, the program becomes more sluggish (being forced to page to disk), other programs running at the same time slow down (because available memory is crowded out), and the operating system may end up killing the program for using too much memory.

There's another problem, too. Look at the second for loop - the one doing the actual work:

```
MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This loop can't even *start* until the *last* square number is calculated. See that? If a human is using this program, it will seem to hang, and they'll think you're a bad programmer.

Python provides a great way to sidestep these problems, called a *generator function*. A generator function looks a lot like a regular function, except instead of “return”, it uses the *yield* keyword. Here's a simple example, generating the numbers 1, 2 3:

```
def gen_numbers():
    value = 1
    while value <= 3:
        yield value
        value += 1
```

You use it like this:

```
for num in gen_numbers():
    print(num)
```

A function is a generator function if and only if it contains a “yield” statement. And it behaves quite differently from a regular function. When you run “gen_numbers()”, it starts at first line, just like a normal function. But when it gets to the “yield” line, it PAUSES.

The generator function produces the value to the consumer - the “for num in gen_numbers()” loop, in this case - but it doesn't really return. “gen_numbers()” just stops, surrendering control of the thread. And when the “for” loop advances to the next loop, and needs a new value, the generator function *un*-pauses. It goes to the next line, “value += 1”, and continues like before. This keeps going until the generator function breaks out of its while loop, and gets to the end of its function body. That signals to the “for” loop that the sequence is done, and your program continues like normal.

Now, you could have “gen_numbers” return [1, 2, 3] and it would behave the same. Except for one HUGE difference: this generator function is **entirely scalable**. Imagine this more flexible version:

```
# Like before, but you can choose how many numbers to generate.
def gen_numbers(limit):
    value = 1
    while value <= limit:
        yield value
        value += 1

# Pass in 3, or 3000, or 3 million.
for num in gen_numbers(3):
    print(num)
```

Imagine passing in 5,000,000,000. A list-based version would create a list that long, filling up memory - the same problem "fetch_squares()" has. But "gen_numbers" is lazy in a good way. "yield" produces each value just in time, when it's needed. And it doesn't calculate the next value until you ask for it. It uses the same amount of memory whether you pass in 3 for limit, or 3,000, or three *trillion*. And it's responsive: you can use the first value right away, and not wait for it to calculate every value in the sequence.

Exercise: Generating Squares

Here's that fetch_squares again:

```
def fetch_squares(root_limit):
    squares = []
    for root in range(root_limit):
        squares.append(root*root)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    print(square)
```

Since it returns a list, it's not very scalable. Let's write a generator function called "gen_squares()" that yields each value, one at a time. Copy the following into a new file, called "gen_squares.py":

```
def gen_squares(max_root):
```

```
    "Write your generator function here."

MAX = 5
for square in gen_squares(MAX):
    print(square)
```

Write code inside “gen_squares()”, so that the program prints out the square numbers from one to five. Remember, “gen_squares()” MUST have a “yield” statement - that's what makes it a generator function.

(If you're not able to write code right now, at least mentally consider how you'd do it. But actually writing code is far better. Do one of these two things before you move on.)

Secret Bottlenecks to Scalable Python

Did you get a working version of “gen_squares()”? There's actually several good ways to do it. Here's one:

```
def gen_squares(root_limit):
    root = 0
    while root < root_limit:
        yield root * root
        root += 1
```

Remember, this is called a "generator function". You know this is a *generator* function because it has “yield” instead of “return”. It's a kind of function, but behaves differently, because it's completely scalable. If you pass in five trillion, it won't create a list of five trillion elements, like the original “fetch_squares()” function did.

I point this out to you because it changes the kind of programmer you are. As you get in the habit of using generator functions as much as you can, your code automatically becomes more scalable. Because you're naturally omitting memory bottlenecks that would otherwise be there. That also tends to make your code more performant. And people recognize how it improves the quality of your code. You build a reputation as someone who can write robust software, that can handle larger, difficult data sets.

Now, there's still something you have to be mindful of in order for this to work. Because there's some subtle ways to hobble your generator functions, if you're not careful. Let's test your ability to spot them.

Exercise: Hidden Scalability Bottlenecks

Here's another version of “gen_squares()” you might write:

```
def gen_squares(root_limit):
    for root in range(root_limit):
        yield root * root
```

In fact, there's a potential barrier to scalability hidden inside here. And even if you see it right away, it's more subtle than you think.

1) What do you think is the problem?
2) And how would you solve it, to make this version of “gen_squares()” more scalable?

Think about this before you continue.

Applying Generators

Did you spot the scalability bottleneck? Again, here's that alternate version of gen_squares:

```
def gen_squares(root_limit):
    for root in range(root_limit):
        yield root * root
```

Here's the tricky part: *it depends on which version of Python you're using.* You see - and pay attention even if you’re only using Python 3, because there’s a larger point here – in Python 2, “range()” returns a *list*:

```
>>> # In Python 2:
... range(5)
[0, 1, 2, 3, 4]
```

But In Python 3, it doesn't do that. Instead, it returns what's called a *range object*. A range object is itself scalable. It's not a generator object; but it's similar, in that it produces its values one at a time, just as they're needed.

(What do you do in Python 2, then? Use “xrange()” instead. That's the scalable version of “range()” in Python 2. In Python 3, “xrange()” was renamed “range()”, and the old returning-a-list “range()” function went away completely.)

Here’s the larger point: When you write a generator function, it's only as scalable as the code you write inside it.

Here's a horrible version of “gen_squares()”:

```
# Don't do this!!
def bad_gen_squares(root_limit):
    squares = []
    for root in range(root_limit):
        squares.append(root*root)
    for square in squares:
        yield squares
```

Do you see how this is completely unscalable? Once you've had a bit of practice, looking at code like this becomes painful, and writing scalable generator functions becomes straightforward. Just be mindful.

Now, our whole theme here is how to really increase your power as a Python developer. So in the next lesson, we're going to switch gears to another very important topic. But first, let's test your understanding of generator functions, and how to apply them.

Exercise: Using Generator Functions

1) What are some ways you can use generator functions in your own code? Think of at least a couple of examples.

2) We've now wrapped up the generator portion of the guide. Are there any eye openers that change how you think about your code?

Answer these questions for yourself before you continue.

There's a great deal more to learn about generator functions - including the massively useful, higher level design patterns that use generators as a foundation. For now, let's switch our focus to collections and data structures.

Key #2: Higher-Level Collections

Now, let's switch gears, to the second key: **expressive collections** and data structures. That's important in any language, isn't it?

This guide is for people who aren't new to Python, so I'll assume you know how to work with Python's lists and dictionaries. Imagine you need a list of objects. Not an iterator – this time, you need an actual list. Again, we'll use squares, just to make it simple (though this all applies to any list):

```
# Create a list of the first five squares.
>>> squares = []
>>> for num in range(5):
...         squares.append(num * num)
>>> print(squares)
[0, 1, 4, 9, 16]
```

This snippet indeed creates a list of the first five square numbers (starting with zero). Would you describe this code as high level, or low level?

It's certainly high level compared to what you would do in a language like C. But in another sense, it's relatively low level. In these three lines, you're telling Python *how* to create what you want.

In a higher level language, you'd be more declarative. You'd say: "Python, this is *what* I want. Figure out how to build it for me."

You get the difference? “How” versus “what”. In fact, Python lets you do this! You can create the same list this way:

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

This is called a **list comprehension**. It's a weird name, but a simple idea. In Python, a list comprehension is a way to create a list. That's all. It's a high-level and *declarative* way to create a new list.

(You may be familiar with simple comprehensions like this already. If so, skip a few paragraphs ahead, to where we cover facets of comprehensions most people don't know about.)

Look inside the square brackets. The list comprehension has this form:

```
[ EXPRESSION for VARNAME in SEQUENCE ]
```

The “for” and “in” keywords are required. So are the square brackets - and that makes sense, because this creates a list. “EXPRESSION” is any Python expression; it's written in terms of the variable. And “SEQUENCE” is any ordered sequence. It can be a list, a generator object, or something else.

Here's another example:

```
>>> [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]
```

The expression field determines the final values in the list you're creating. It can be a function call on the variable, or even invoke a method of the object:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]

>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> [ pet.upper() for pet in pets ]
['DOG', 'PARAKEET', 'CAT', 'LLAMA']
```

Notice the final list has the same number of elements as the source sequence. But Python list comprehensions also let you filter out certain elements, keeping only those that meet certain criteria. You do that by adding an “if” clause:

```
>>>> numbers = [ 9, -1, -4, 20, 11, -3 ]

>>> # Positive numbers:
... [ x for x in numbers if x > 0 ]
[9, 20, 11]

>>> # Squares of even numbers:
>>> def iseven(number):
...     return number % 2 == 0
>>> [ x**2 for x in numbers if iseven(x) ]
[16, 400]
```

The structure is:

```
[ EXPR for VAR in SEQ if CONDITION ]
```

Here, “condition” is something that evaluates to either True or False. If True, the item is kept in the final list; if not, it's skipped over and left out. It can be a function or method call, or some other expression in terms of the variable.

That's the basics of list comprehensions. Before you learn some more advanced usage - including how comprehensions tie back to generators! - let's test your understanding and write some code.

Exercise: List Comprehensions

Create a new Python file named "lcpractice.py", and paste in the following:

```
colors = ["red", "green", "blue"]
numbers = [ 9, -1, -4, 20, 11, -3 ]
```

Below that in the file, write Python code that uses them in list comprehensions to produce the following lists:

```
["RED", "GREEN", "BLUE"]
[9, 20, 11]
[ 9, 1, 4, 20, 11, 3 ]
```

(Hints: for the first, look up the methods of string. And for the last one, use a built-in function that starts with the letter "a".)

Beyond List Comprehensions

Let's look at some advanced features of comprehensions, which not many people know about, before moving on to the next topic. First, your list comprehensions can have several “for” and “if” clauses:

```
colors = ["orange", "purple", "pink"]
toys = ["bike", "basketball", "skateboard", "doll"]
gifts = [
    color + " " + toy
    for color in colors
    for toy in toys
    ]
```

Then “gifts” contains:

```
[['orange bike', 'orange basketball', 'orange skateboard',
 'orange doll', 'purple bike', 'purple basketball',
 'purple skateboard', 'purple doll', 'pink bike',
 'pink basketball', 'pink skateboard', 'pink doll']
```

And for multiple “if”'s:

```
numbers = [ 9, -1, -4, 20, 11, -3 ]
positive_evens = [
    num for num in numbers
    if num > 0
    if num % 2 == 0
    ]
```

The “if” clauses are and-ed together. In other words, a number is kept only if it meets BOTH criteria:

```
>>> print(positive_evens)
[20]
```

So “numbers” had five elements, but only one passes both “if” clauses. This doesn't modify “numbers”, by the way. A list comprehension always creates a new list. It doesn't modify the source sequence (or sequences) in any way.

The benefits of list comprehensions are mainly about readability and maintainability. But there's more than that. The declarative style of a comprehension is so remarkably clear, it actually brings cognitive benefits. Once you're familiar, you'll find code using list comprehensions easier to reason about.

Now, let's talk about generator functions again. You might be wondering: what if you want to write a list comprehension, but need something more scalable? Python actually lets you write generator comprehensions! And the syntax is very similar. All you have to do is write parenthesis instead of square brackets:

```
>>> squares = ( num * num for num in range(5) )
>>> type(squares)
<class 'generator'>
>>> for square in squares:
...         print(square)
0
1
4
9
16
```

This creates a generator *object* directly, without needing a generator *function*. In fact, the code right above is EXACTLY equivalent to defining a generator function, and calling it once:

```
# Typing all this...
def gen_squares(root_limit):
      for root in range(root_limit):
            yield root * root

squares = gen_squares(5)

# ... is EXACTLY like typing just this:
squares = ( num * num for num in range(5) )
```

Usually we define a function or method because we want to reuse it - to call it in several places. Sometimes I'll also define a function I intend to use once, simply because it improves readability of the code. But it's annoying, and *decreases* readability, when you're forced to write a function even if you don't feel you need to. Generator comprehensions let you sometimes avoid that, for simple generator functions.

That wraps up the section on comprehensions. Before we go to the next big topic - Pythonic object-oriented programming - let's test your understanding and deepen what you've learned.

Exercise: Advanced Comprehensions

1) What are some situations where you might use comprehensions - either list comprehensions, generator comprehensions, or another kind?

2) When might you want to use a list comprehension instead of a generator comprehension? And the other way - when might using a generator comprehension be better?

Think about your answers to these questions before you read on.

Key #3: Pythonic Design Patterns

Now we're ready for the third key: **Pythonic OOP**. Because there are many, *many* books and articles on design patterns…

But nearly ALL of them are written for languages like Java, C++, and C#.

And while most of these patterns apply to Python, they apply *differently*. Because as a language, Python's feature set is different.

This is a deep, powerful, and immensely rewarding topic. So we'll devote the rest of this Playbook to exploring it.

First, let's learn about properties. You may be familiar with this from other languages; we're talking about getters and setters. The basic idea: when your objects have member variables, you want to have code execute when that member variable is accessed.

Why, exactly? Basically, it's the foundation of MANY useful design patterns, especially in Python.

Let's start simple. Here's a Person class:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
```

```
    @property
    def full_name(self):
        return self.first + " " + self.last
```

The constructor is straightforward; it just takes the first and last name. The full_name method is more interesting, because of what's on the previous line: “@property”. It turns out “property” is a built-in Python *decorator*. What's a decorator? A decorator adds behavior around a function or method. You apply it when you *define* that method, and you do that by writing at-sign followed by the decorator name, on the previous line. So when you write “@property” followed by “def full_name”, we say you are *decorating* the full_name method.

Okay, fine. So what does this actually do? Because it looks like you've defined a method called “full_name”. But it doesn't act like a method:

```
>>> guy = Person("John", "Doe")
>>> print(guy.full_name)
John Doe
```

This “full_name” attribute acts like a member variable! In fact, you can't call it like a method, even if you want to:

```
>>> guy.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

In standard OOP terms, you've just created a *getter*. Python does it differently than other languages, because it looks like a member variable from the outside. But it's implemented as a method. Notice how the object's data is denormalized. You have two member variables, “self.first” and “self.last”. And “full_name” *appears* to be a member variable, but it calculates its value dynamically, at run-time:

```
>>> guy.full_name
```

```
'John Doe'
>>> guy.first = "Jim"
>>> guy.full_name
'Jim Doe'
```

When you access “guy.full_name”, that causes the full_name *method* to be called.

By default, Python properties are read only:

```
>>> guy.full_name = "Tom Smith"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Is this a feature, or a bug? It depends. For some design patterns, this is exactly what you want. You might want to create a read-only member variable, for example. And sometimes, you need to be able to assign to the property. To do this, you define a setter, like this:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return self.first + " " + self.last

    @full_name.setter
    def full_name(self, new_name):
        first, last = new_name.split()
        self.first = first
        self.last = last
```

That lets you assign to it:

```
>>> guy.full_name = "Tom Smith"
>>> print(guy.first)
```

```
Tom
>>> print(guy.last)
Smith
```

Here's another way of saying this: when you create a property, using "@property", that automatically creates a getter, but *not* a setter. To also create a setter, you first create the property, and then create the setter using "@full_name.setter". (Or whatever the name of the property is.) So the newly created property has a "setter" attribute, which you use to decorate the setter method.

You might be having questions about how this all works: why you define full_name twice, what the "full_name.setter" is about, and so on. There's a reason for all of it, but practically speaking, you don't need to know how it works in order to use it.

Let's put this into practice, and have you write code for your own properties.

Exercise: Python Properties

Let's write code for a Thermometer class. Instances will have "celsius" and "fahrenheit" attributes, but only one of them is an actual member variable; the other is a property. Start by copying the following lines into a blank Python file:

```
class Thermometer:
    def __init__(self, celsius):
        self.celsius = celsius
    # Add your properties for "fahrenheit" here

thermo = Thermometer(15)
print(thermo.fahrenheit)
thermo.fahrenheit = 68
print(thermo.celsius)
```

Your job is to add a getter and setter to the Thermometer class, for the fahrenheit property. When you run the program, it ought to print out

```
59.0
20.0
```

Notice the values are rounded. Some hints:

- Convert Fahrenheit to Celsius and back using the formulas "F = (9/5)*C + 32" and "C = (5/9)*(F-32)".
- Make your getter and both round the calculated value to the nearest degree, using Python's built-in round() function.

Pythonic Refactoring With Properties

Now, about the last exercise - the Thermometer class. Here's one good solution:

```
class Thermometer:
    def __init__(self, celsius):
        self.celsius = celsius
    # Add your properties for "fahrenheit" here
    @property
    def fahrenheit(self):
        return 32 + (9/5)*self.celsius
    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (5/9)*(value - 32)
```

Now, we've just covered the basics of properties so far. Like I said, they underlie many useful design patterns. While we don't have room to cover them all in this guide, I HAVE to show you one more wonderful thing about how Python does properties.

Imagine you write a Money class:

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

```
    # Plus some methods for financial calculations.
```

This class works for currencies based on dollars: the USA, New Zealand, Singapore, etc. It's packaged in a library that turns out to be so useful, other developers start using it too - some even for different projects.

Now, one day, when adding a new method, you realize many of the financial calculations will be a lot simpler if you simply store the amount as the total number of cents, rather than dollars and cents separately. So you refactor:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = cents + 100 * dollars
    # And refactor all other methods to use total_cents.
```

And you also change your code to use the new member variable, instead of dollars and cents. Now, this works fine. Except it creates a new problem. Can you think of what it is?

Here's the issue: what about the other developers? Your Money class started off with public “dollars” and “cents” variables, and other developers just started using them. Now, you're changing the class in such a way that all their code breaks. And when their code breaks, you're the one who gets blamed for it.

Now, if not much code is using this class, AND you have permission to modify all the code that does, maybe you can just change them all together. But that's not always possible, or practical. The situation is so messy, you may need to abandon the change altogether.

But with Python, you have another choice... using properties! Go ahead and make the change - You go ahead and add “total_cents”, and get rid of the “dollars” and “cents” member variables. And to handle existing code, you also define getters and setters for the last two:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = 100 * dollars + cents
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents = 100 * new_dollars + self.cents
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents = 100 * self.dollars + new_cents
```

The wonderful thing about this: the other developers using the Money class don't need to change their code AT ALL. It just continues to work with your new version of the class. In fact, they don't even have to know you changed it! This refactoring benefit is one of the great things about how Python does properties.

To wrap up this section on properties, let's test your ability to use them during refactoring.

Exercise: Refactoring With Properties

Copy and paste this into a new Python file:

```
class Thermometer:
    def __init__(self, kelvin):
        self.kelvin = kelvin
    @property
    def fahrenheit(self):
        return 32 + (9.0/5) * (self.kelvin - 273.15)
    @fahrenheit.setter
    def fahrenheit(self, value):
        self.kelvin = 273.15 + (5.0/9)*(value - 32)
    # Add your properties for "celsius" here
```

```
thermo = Thermometer(288.0)

print(round(thermo.kelvin, 1))
print(round(thermo.celsius, 1))
thermo.fahrenheit = 68
print(round(thermo.kelvin, 1))
print(round(thermo.celsius, 1))
```

When you run this little program, you'll get an error. Fix it by adding a getter and setter for celsius. Hint: The formula relating temperature in degrees Kelvin (K) to degrees Celsius (C) is "K = C + 273.15". You'll know it's working when you get the following output:

```
288.0
14.9
293.1
20.0
```

The Pythonic Factory Pattern

Great job getting this far! You're in the home stretch of this guide. Before we go to the next topic, here's the full code of the working Thermometer class. You can skip to the “celsius” property at the end - its getter and setter are the only things that were added:

```
class Thermometer:
    def __init__(self, kelvin):
        self.kelvin = kelvin
    @property
    def fahrenheit(self):
        return 32 + (9.0/5) * (self.kelvin - 273.15)
    @fahrenheit.setter
    def fahrenheit(self, value):
        self.kelvin = 273.15 + (5.0/9)*(value - 32)
    # Just add the getter and setter for "celsius"
    @property
    def celsius(self):
        return self.kelvin - 273.15
    @celsius.setter
    def celsius(self, value):
```

```
        self.kelvin = value + 273.15
```

Now, onward. Let's talk about factories.

There are several different design patterns with the word "factory" in their name. But let's focus on the “simple factory” pattern - which is the most basic, and most useful, kind of factory. In short, a factory is a function that helps you create an object. You use it when either the constructor is quite complicated, or you need to do some pre-calculations before you can even create the object at all. Let's work with the original money class again:

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

We're rolling back to the original version, and this time we're going to focus on evolving the outer interface. Let me explain. The constructor takes two arguments, dollars and cents. And imagine that in several places in your code, you're given the amount of money as a string - something like "$145.08" - and you need to parse it before you can instantiate Money. Since you need to do it several times, you wrap it in a function:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError('Invalid amount: {}'.format(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This is a factory function for the Money class. It takes the data you *have*, uses it to calculate the arguments you *need*, and creates that money object for you. The code isn't super hard, but it's not quite trivial, and you definitely want to wrap it in a reusable function like

this. It'll raise a ValueError if passed an invalid string, and otherwise gives you back a Money object.

That's all great. But this factory function has one major problem, mainly affecting larger applications, but which also can affect small scripts. Look at the code for “money_from_string()”. Do you see the issue?

The big problem is this: what if you need to subclass Money? This factory function ONLY works with the Money class itself. Imagine creating a TipMoney class, to keep track of tips in a restaurant chain. You'd have to make a different factory function for each subclass.

Python provides a much better solution: the *class method*. You define one using the @classmethod decorator:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_string(cls, amount):
        import re
        match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        if match is None:
            raise ValueError('Invalid amount: {}'.format(amount))
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

Look carefully at this new method, “from_string()”. On the previous line, you write “@classmethod”. That changes the nature of the method: look at its first argument. It's not “self”, it's “cls”, short for "class". The code in the method itself is similar to the first factory function, except the very last line. Read it carefully - do you see what it's doing?

It turns out that “cls” is a variable holding the current class. Because in Python, everything is an object - EVERYTHING. If something is an

object, you can store it in a variable. And when you write “class Money”, creating the class named Money, *that class is itself an object*.

Python automatically puts that class in the “cls” variable. Here's how you use the class method:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

```
>>> piggie_bank_cash = Money.from_string("$14.72")
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
14
>>> piggie_bank_cash.cents
72
```

Notice that “from_string()” is invoked off the Money class directly. Not an *instance* of the class, like most methods. Get that difference? And notice that “piggie_bank_cash” is of type Money.

This is already an improvement, because the “from_string()” class is now housed inside the Money class. The old “money_from_string()” function was an unattached function, so this is more organized.

But the REAL benefit comes when you subclass:

```
>>> class TipMoney(Money):
...     "A class representing tips at a restaurant."
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

This factory function automatically works with subclasses! You just need to write the class method once, in the base class.

You may know about static methods in other languages. That's quite different from class methods: for one thing, static methods have always had problems with inheritance, which class methods handle quite nicely.

Let's test your understanding of how to use class methods before we move on.

Exercise: Class Methods

What are some recent situations where you would have benefited from using class methods - or times in the near future you'd expect them to be useful? Come up with at least three examples.

What's Next...

In Part 2, we looked at a few intermediate Python programming topics…

But if you REALLY want to excel as a developer using Python… there's a lot more to learn…

My hope is that this playbook will give you clarity, and give you a better idea of what REALLY works... breaking you through to that level of Python mastery you've always wanted, but has been frustratingly just out of reach.

This system eliminates all of the guesswork in getting in that top 1% of "elite" Python developers, no matter what you're using Python for... and stay solidly ahead of the curve for years to come.

If you'd like to speak with us on how we can help you use this powerful system, we'd love to talk to you.

There is never any pressure on our clarity sessions. Our objective is to figure out where you're having the most trouble breaking through as a Python coder, and then help you figure out what the best plan of action is...

And if you'd like to set up a clarity session with us, follow the URL below to do that:

powerfulpython.com/apply

Feel free to send us an email (if you need anything) or if you just want to say 'thanks for the Playbook'.
(service@powerfulpython.com)

Looking forward to talking soon!

Aaron Maxwell
Founder, Powerful Python