Looping Like a Pro in Python

David "DB" Baumgold
@singingwolfboy

Shameless Plug

I'm DB!

- Freelance web developer
- Corporate trainer
- Serial conference presenter
- Friendly & helpful

Hire me!

davidbaumgold.com

What is a Loop?

A loop is a
method of programming
that allows you to
repeat the same action or idea
over and over,
without repeating code.

Without Loops

Display the letter P.
Display the letter Y.
Display the letter T.
Display the letter H.
Display the letter O.
Display the letter N.

With Loops

Display each letter in the word "PYTHON".

Vocabulary

Iterable: anything you can loop over (list, string, dictionary, etc)

Iteration: one execution of the loop

Index: a variable that keeps track of which iteration you're currently on

Vocabulary

Truthy: any value in Python that is either `True`, or coerces to `True`. Examples: `1, "abc", ["foo"]`

Falsy: any value in Python that is either `False`, or coerces to `False`. Examples: `0, "", [], None`

Loops in Python

There are **two ways** to do loops in Python:

`for` and `while`

Use `for` when you know exactly how many times you need to loop.

Use `while` when you need to keep going until you figure out it's time to stop.

while

colon after conditional

```
while conditional:
    body
    (can be multiple lines)
```

fancy word for “test”

The body is indented!

while

```
arrived = False

while not arrived:
    print("Are we there yet?")
    rand = random.random()
    arrived = rand < 0.1

print("We're here!")
print("When do we leave?")
```

conditional

`while`

1. Check the conditional
2. If it's truthy, evaluate the body, then go back to step 1
3. Otherwise, skip the body and move on with the program

while

```
task = get_task()

while task:
    execute(task)
    task = get_task()
```

What if there are no tasks available?

while

infinite loop!

```
while True:
    task = get_task()
    if task:
        execute(task)
    else:
        time.sleep(1)
```

Loop Control Statements

There are **two ways** to modify loop controls in Python:

`continue` and `break`

Use `continue` to skip the rest of the body,
and go on to the next iteration of the loop.

Use `break` to end the loop entirely,
and move on with the program.

Loop Control Statements

```
while True:
    task = get_task()
    if should_skip(task):
        print("No thanks")
        continue
    execute(task)
```

This task is skipped, but the worker will continue executing other tasks

Loop Control Statements

```
while True:
    task = get_task()
    if task.id == 13:
        print("I quit!")
        break
    execute(task)
```

This worker
won't execute
any more tasks!

Review: `while` loop

```
while conditional:
    body
```

Execute the **body** over and over
until **conditional** is falsy.

`continue`: skip the rest of the body,
go on to the next iteration.

`break`: end the loop entirely,
go on with the program.

for

colon after iterable

```
for var in iterable:
    body
    (can be multiple lines)
```

The body is indented!

for

```
for num in [1, 2, 3]:
    print(num)
```

var

iterable

for

```
for num in [1, 2, 3]:
    print(num)
```

- Set `num` to `1`
- `print(1)`
- Set `num` to `2`
- `print(2)`
- Set `num` to `3`
- `print(3)`

`for`

1. Try to get the next value from the iterable

If there *is* a next value…

Oops, the iterable is empty!

2. Set the variable to that value

3. Execute the body

4. Go back to step 1

5. Skip the body and move on with the program

Loop Control Statements

`continue` and `break`

work with `for` loops, as well!

Use `continue` to skip the rest of the body,
and go on to the next iteration of the loop.

Use `break` to end the loop entirely,
and move on with the program.

Loop Control Statements

```
for task in todo_list:
    if should_skip(task):
        print("No thanks")
        continue
    execute(task)
```

This task is skipped, but the worker will continue executing other tasks

Loop Control Statements

```
for task in todo_list:
    if task.id == 13:
        print("I quit!")
        break
    execute(task)
```

This worker
won't execute
any more tasks!

Loop Indexes

Index: a variable that keeps track of which iteration you're currently on

```
foods = ["bread", "milk", "eggs"]

i = 0
while i < 3:
    food = foods[i]
    print(food)
    i = i + 1
```

DON'T DO THIS

Loop Indexes

Don't use an index variable if you don't need it!

```
foods = ["bread", "milk", "eggs"]

for food in foods:
    print(food)
```

“But I need the list index
in the body of the loop!
Guess I’ll do it like C does…”

NOPE

Loop Indexes

```
foods = ["bread", "milk", "eggs"]

for i, food in enumerate(foods):
    print(str(i) + ": " + food)
```

Result:

```
0: bread
1: milk
2: eggs
```

Loop Indexes

```
foods = ["bread", "milk", "eggs"]

for i, food in enumerate(foods):
    print(str(i) + ": " + food)
```

???

What is this sorcery??

Sidebar: Tuple Unpacking

```
>>> x, y = (2, 3)
>>> x
2
>>> y
3
```

Multiple assignment in Python:
assign values based on position in tuple

Sidebar: Tuple Unpacking

```
>>> for x, y in [(1, 2), (3, 4)]:
...     print(x)
...     print(y)
...
1
2
3
4
```

It works in `for` loops, too!

`for`

1. Try to get the next value from the iterable

If there *is* a next value…

Oops, the iterable is empty!

2. Set the variable to that value

3. Execute the body

4. Go back to step 1

5. Skip the body and move on with the program

`enumerate()`

`enumerate()` is a built-in function that puts an index in each item of a list

```
>>> foods
[
  "bread",
  "milk",
  "eggs",
]
```

```
>>> enumerate(foods)
[
  (0, "bread"),
  (1, "milk"),
  (2, "eggs"),
]
```

`for` with index

```
foods = ["bread", "milk", "eggs"]

for i, food in enumerate(foods):
    print(str(i) + ": " + food)
```

- Set `i` to `0`, `food` to `"bread"`
- print `"0: bread"`
- Set `i` to `1`, `food` to `"milk"`
- print `"1: milk"`
- Set `i` to `2`, `food` to `"eggs"`
- print `"2: eggs"`

Useful Built-in Functions

- `enumerate()`: adds indexes to list values
- `sorted()`: returns a sorted copy of the list
- `reversed()`: iterates over the list *backwards*
- `range()`: iterates over numbers
- `zip()`: combines lists together

sorted()

```
>>> foods = ["bread", "milk", "eggs"]
>>>
>>> for food in sorted(foods):
...     print(food)
...
bread
eggs
milk
```

reversed()

```
>>> foods = ["bread", "milk", "eggs"]
>>>
>>> for food in reversed(foods):
...     print(food)
...
eggs
milk
bread
```

range()

```
>>> for num in range(5):
...     print(num)
...
0
1
2
3
4
```

“I need to loop over two lists simultaneously!
Guess I need to go back to the C indexing method…”

NOPE

Looping Over Multiple Lists

```
people = ["alice", "bob", "carol"]
foods = ["milk", "eggs", "bread"]

i = 0
while i < len(people):
    person = people[i]
    food = foods[i]
    print(person + " likes " + food)
    i = i + 1
```

DON'T DO THIS

Looping Over Multiple Lists

```
people = ["alice", "bob", "carol"]
foods = ["milk", "eggs", "bread"]

for person, food in zip(people, foods):
    print(person + " likes " + food)
```

Result:

```
alice likes milk
bob likes eggs
carol likes bread
```

Looping Over Multiple Lists

`zip()` is a built-in function
that combines multiple lists like a zipper

```
>>> people
[
  "alice",
  "bob",
  "carol",
]
```

```
>>> foods
[
  "bread",
  "milk",
  "eggs",
]
```

```
>>> zip(people, foods)
[
  ("alice", "bread"),
  ("bob", "milk"),
  ("carol", "eggs"),
]
```

It even works with more than two lists at a time!

Review: `for` loop

```
for var in iterable:
    body
```

Set **var** to next item from **iterable**, execute **body**, do it again until **iterable** is empty

It's useful to modify **iterable** with functions

Useful Built-in Functions

- `enumerate()`: adds list indexes
- `sorted()`: returns a sorted copy of the list
- `reversed()`: iterates over the list *backwards*
- `range()`: iterates over numbers
- `zip()`: combines lists together

Nested Loops

Sometimes you need a loop inside another loop:

```
for person1 in group:
    for person2 in group:
        person1.greet(person2)
```

Nested Loops

They can get complicated…

```
for person1 in group:
    for person2 in group:
        if person1 == person2:
            continue
        person1.greet(person2)
```

Nested Loops

They can get complicated…

```
for person1 in group:
    if person1.is_antisocial():
        continue
    for person2 in group:
        if person1 == person2:
            continue
        person1.greet(person2)
```

Nested Loops

What if you want to `break` in the middle?

```
for person1 in group:
    if person1.is_antisocial():
        continue
    for person2 in group:
        if person1 == person2:
            continue
        if bad_match(person1, person2):
            break
        person1.greet(person2)
```

breaks out of the inner loop,
but not the outer one!

Nested Loops

How do we improve these complicated nested loops?

Separate iteration logic from loop body!

How do we do that?

Generators!

Generators

```
>>> def grocery_list():
...     yield "milk"
...     yield "eggs"
...     yield "bread"
...
>>> for food in grocery_list():
...     print(food)
...
milk
eggs
bread
```

Generators

```
def pairs(group):
    for person1 in group:
        for person2 in group:
            yield person1, person2

for person1, person2 in pairs(group):
    person1.greet(person2)
```

Generators

```
def pairs(group):
    for person1 in group:
        for person2 in group:
            if person1 == person2:
                continue
            yield person1, person2

for person1, person2 in pairs(group):
    person1.greet(person2)
```

Generators

```
def pairs(group):
    for person1 in group:
        if person1.is_antisocial():
            continue
        for person2 in group:
            if person1 == person2:
                continue
            yield person1, person2

for person1, person2 in pairs(group):
    person1.greet(person2)
```

Generators

```
def pairs(group):
    for person1 in group:
        if person1.is_antisocial():
            continue
        for person2 in group:
            if person1 == person2:
                continue
            if bad_match(person1, person2):
                return
            yield person1, person2

for person1, person2 in pairs(group):
    person1.greet(person2)
```

Generators

Generators are evaluated lazily: only when requested. You may break out of the loop before the code is evaluated!

```
$ python3 -m timeit 'x = range(1000000)'
1000000 loops, best of 3: 0.357 usec per loop
$ python3 -m timeit 'x = list(range(1000000))'
10 loops, best of 3: 40 msec per loop
```

10,000x slower!

list

generator

Generators

```
users = User.query.all()

for i, user in enumerate(users):
    print(user)
    if i == 10:
        break
```

If `users` is a generator, this code will only load the first 10 users.
If it's a list, it will load *all* the users, and throw away all but the first 10!

Infinite Generators

```
def road_trip():
    while True:
        yield "Are we there yet?"

for whine in road_trip():
    print(whine)
```

If you like `for` better than `while`,
you can write infinite generators!
(Believe it or not, this is occasionally useful.)

Infinite Generators

```
def get_widgets():
    page = 1
    while True:
        url = "/widgets?page={}".format(page))
        resp = requests.get(url)
        for widget in resp.json()["widgets"]:
            yield widget
        if resp.json()["hasNextPage"]:
            page = page + 1
        else:
            return
```

Infinite Generators

```
for widget in get_widgets():
    process(widget)
```

This will automatically make HTTP requests
in the background, whenever necessary,
leaving you to focus on the logic of your program!

Only loads the widgets you actually use.
If you only use the first 10 widgets,
it won't request the rest!

Review: Generators

- Useful for refactoring nested loops: separate iteration logic from loop body!
- Performance improvements: only execute code when necessary!
- Infinite generators: occasionally useful!

Advanced Looping: `itertools`

`itertools` is a module in the Python standard library that provides even more useful functions for iterating over loops

```
import itertools
```

Advanced Looping: `itertools`

Need permutations and combinations?
`itertools` has got you covered.

```
pairs = itertools.combinations(group)

for person1, person2 in pairs:
    person1.greet(person2)
```

no repeated greetings!

Advanced Looping: `itertools`

Need to easily group values by category?
`itertools.groupby` is very handy:
it's like SQL `GROUP BY` queries,
but for unaggregated information.

This is great for displaying categorized data from a database, like you might do with Django!

Advanced Looping: `itertools`

For example, let's say you need to display
three lists of student names, categorized by grade level

itertools.groupby

```
students = (
  Student.query
  .order_by('grade_level')
  .all()
)

s_by_grade = itertools.groupby(
  students,
  lambda s: s.grade_level,
)
```

itertools.groupby

`s_by_grade` looks like this:

```
[(1, [<student>, <student>, <student>]),
 (2, [<student>, <student>, <student>]),
 (3, [<student>, <student>, <student>]), ]
```

grade level

list of students in that grade level

itertools.groupby

```
for grade, students in s_by_grade:
    print("Students in grade " + grade)
    for student in students:
        print(student.name)
```

Any questions?

- `while` loops
- `for` loops
- loop control statements
- functions that modify iterators
- nested loops
- generators
- `itertools`

David Baumgold // @singingwolfboy