Exceptions

EAFP vs LBYL

→ when we think something unexpected may go wrong in our code

→ figure out if something is going to wrong before we do it

→ LBYL Look Before You Leap

→ just do it, and handle the exception if it occurs

→ EAFP Easier to Ask Forgiveness than Permission

generally in Python → follow EAFP

→ exception handling

Why EAFP?

Something that is exceptional should be infrequent

→ if we are dividing two integers in a loop that repeats 1,000 times

→ out of every 1,000 times we run, we expect division by zero to occur 5 times

LBYL → test that divisor is non-zero 1,000 times

EAFP → just do it, and handle the division by zero error 5 times

→ often more efficient

→ also trying to fully determine if something is going to go wrong is a lot harder to write than just handling things when they do go wrong

with LBYL we need to take many scenarios into account.

General Suggestions for Exception Handling

→ in general we do not want to just handle any exception anywhere in our code

→ too much work

→ cannot anticipate every point of failure

→ it's OK for program to terminate - we can figure out what went wrong and attempt to fix it later – possibly handling that case specifically

→ if we don't know exactly why or where the problem occurs in our code, there's not much we can do to recover from the exception

→ we handle exceptions that are raised by small chunks of code

→ we try to handle very specific exceptions, not broad ones

→ usually handle exceptions that we can do something about

We should make our exceptions very specific

LBYL

```
l = [1, 2, 3, 4, 5]

while len(l) > 0:
    print(l.pop())

print('all done')
```

```
5
4
3
2
1
all done
```

len(l) is executed len(l) times

EAFP

```
l = [1, 2, 3, 4, 5]

try:
    while True:
        print(l.pop())
except IndexError:
    # index error means list is now empty - expected behavior
    print('All done - all elements have been popped.')

print('code resumes here...')
```

```
5
4
3
2
1
All done - all elements have been popped.
code resumes here...
```

- there is no len(l) here!
- Notice that we use a specific exception

What if we don't use a specific exception?

```
l = (1, 2, 3, 4, 5)

try:
    while True:
        print(l.pop())
except Exception:
    # what happened? IndexError? something else?
    print('All done - all elements have been popped.')

print('code resumes here...')
```

```
All done - all elements have been popped.
code resumes here...
```

```
l
```

```
(1, 2, 3, 4, 5)
```

```
data = [10, 20, 'abc']

sum_data = 0
count_data = 0

try:
    for element in data:
        try:
            sum_data += element
            count_data += 1
        except TypeError:
            # skip
            pass
    average = sum_data / count_data
except ZeroDivisionError:
    average = 0

print(f'average = {average}')
```

```
average = 15.0
```

Nested exceptions

Iterables / Iterators

Read both lecture slides and notebooks for iterators and generators

Functions

unpack arguments

```
def product(*values):
    print(values)
    prod = 1
    for value in values:
        prod *= value
    return prod

l = [1, 2, 3, 4]

product(*l)
```

```
(1, 2, 3, 4)

24
```

How to force keyword-only arguments

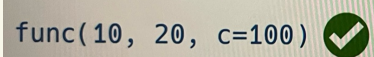
```
def func(a, b, *, c):
    ...
```

→ a and b are positional parameters

→ there are no more positional parameters after that

→ so c is a keyword-only argument

```
func(10, 20, c=100)
```



```
func(10, 20, 30, c=100)
```

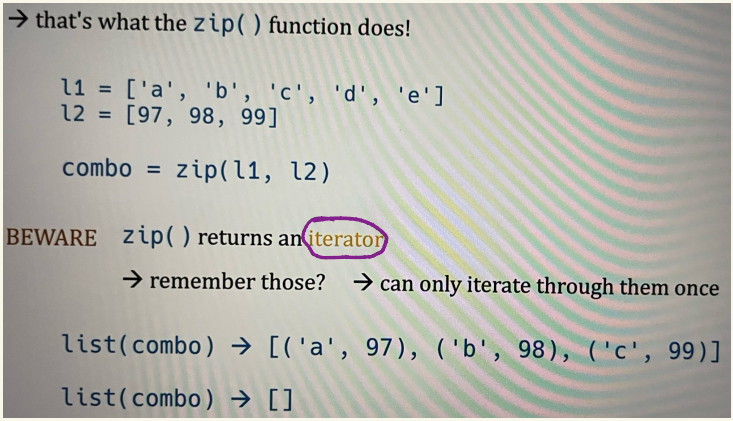
Arbitrary Number of Keyword-only Parameters

→ saw * for arbitrary number of positional arguments

→ use ** for arbitrary number of keyword-only arguments

```
func(a, b, *args, c, d, **kwargs)
```

→ a and b are positional
→ c and d are keyword-only
→ extra positional arguments are scooped up into `args`
→ extra named arguments are scooped up into `kwargs`



Don't forget lambdas!

High-Order Functions

map() function

→ the `map()` function calls a specified function for every element of some iterable

→ very similar to doing something like this:

```
def my_map(func, iterable):
    result = [func(element) for element in iterable]
    return result
```

→ here we are creating a `list` that contains the function `func` applied to every element of `iterable`

→ but it creates a `list`

→ can take a lot of space if iterable is large

→ especially wasteful if we don't iterate over all the values

→ `map()` returns an iterator

```
iterator = map(func, iterable)
```

as we iterate over that iterator:

→ Python moves to the next item in `iterable`

→ calls `func(element)`

→ returns the result

→ less wasted space

→ saves computations if we don't iterate over the whole list

→ equivalently we could also just use a generator expression

```
(func(el) for el in iterable)
```

Use it if the iterable contains a large number of elements!

filter()

Similar is the case for **filter** function

lazy_iterator = **filter**(predicate, iterable)

predicate: function that returns True or False

equivalent generator ⇓

(predicate(el) for el in iterable)

Don't use comprehension if the iterable contains many elements!

sorted() → check lecture slides and notebooks

similar for **min()**, **max()**

sorted(iterable, func)

- returns an iterable (not iterator!)
- func: must return a value for each iterable element it's the key by which the iterable will be sorted

Closures

```
def outer(a, b):
    c = 100

    def inner():
        …

    return inner
```

→ inner can "see" those variables

→ it even retains these values when it is returned

→ the inner function can "capture" those variables

→ this is called a closure

If the inner does not capture any value from the outer function, it does not constitute a closure!

Read the lecture slides on closures for more info.

Example

without closure

```
def time_it(func, *args, **kwargs):
    start = perf_counter()
    result = func(*args, **kwargs)
    end = perf_counter()
    print(f'elapsed: {end - start}')
    return result

result = time_it(factorial, 10_000)

 elapsed: 0.017552107000028627

result = time_it(diagonal_matrix, 10, 10, diagonal=-1)
```

with closure

```
def time_it(func):
    def inner(*args, **kwargs):
        start = perf_counter()
        result = func(*args, **kwargs)
        end = perf_counter()
        print(f'elapsed: {end - start}')
        return result
    return inner
```

```
timed_fact = time_it(factorial)
```

```
timed_diagonal = time_it(diagonal_matrix)
```

```
result = timed_fact(5)
```

```
 elapsed: 2.0769998627656605e-06
```

```
result
```

```
   120
```

Using a closure, it is more natural to call *time_fact(5)* instead of *time_it(factorial, 5)*

Read/Write Files

```
with open(<file_path>, <mode>) as f:
    ⋮
```

it's an iterator
so I can call next(f)

when writing to a file, I need to explicitly write the "new line character"

e.g. I can use the join() method in case I have a list of strings:

```
f.write("\n".join(data))
```

```
def transform_row_for_output(row):
    row = row.strip()
    dt_str, rate = row.split(',')
    year, month, day = split_date(dt_str)

    try:
        float(rate)
    except ValueError:
        return None

    month = str(int(month))
    day = str(int(day))

    result = ','.join([year, month, day, rate])
    result += '\n'
    return result
```

check if it is a float

```
def transform_file(source_file, target_file):
    with open(source_file) as source:
        with open(target_file, 'w') as target:
            next(source)
            target.write('YEAR,MONTH,DAY,EXCH\n')

            for row in source:
                target.write(transform_row_for_output(row))
```

here we are reading one line and writing it directly

'source' is an iterator!

Modules

`import math` → The module is loaded from disk, compiled, and Python assigns a reference 'math' to it → `math.sqrt(4)`

`from math import sqrt`

↓ here I don't have a reference 'math' to math module, but the module is loaded from disk and compiled
e.g. I cannot write
`math.abs()`

↘ I can reference it directly:
`sqrt(4)`

Statistics Module

`import statistics as stats`

`stats.mean(<iterable>)`
`stat.fmean(<iterable>)` → returns always a float

FASTER!!

Practice Sites

→ there are many sites where you c

edabit.com

coderbyte.com

www.codewars.com

www.hackerrank.com

Additional Resources

stackoverflow.com
→ if you have a coding question
→ you probably weren't the first with that question
→ you will probably find an answer, at least close
→ if not, you can post your question
→ just browse questions/answers – incredibly informative

Python Cookbook, by Beazley and Jones (O'Reilly Press)

Fluent Python, by Ramalho (O'Reilly Press)

YouTube → experts such as Hettinger, Beazley, Martelli, PyCon talks

Twitter → Raymond Hettinger (@raymondh)