Exceptions & More on Functions

Carol V. Alexandru-Funakoshi

November 7, 2023

1 About the Midterm

When will the results be published?

When they're ready!

2 Recap of *abstract* and inhertiance

What you need to know about *abstract* classes (in Python):

- *abstract* classes are classes with at least one abstract method
  - An *abstract* class cannot be instantiated
  - By annotating a method with `@abstractmethod` we tell Python that any concrete subclasses must implement that method
- An abstract class should eventually inherit from `ABC`

[196]:
```python
from abc import ABC, abstractmethod

class Document(ABC):
    def __init__(self, name, data):
        self.name = name
        self.data = data

    @abstractmethod
    def draw(self):
        pass

class TextDocument(Document):
    def draw(self):
        lines = self.data.splitlines()
        longest_line = max(len(l) for l in lines)
        res = " " + " " * (longest_line + 2)             # https://www.w3.org/TR/
 ↪xml-entity-names/025.html
        for line in lines:
            res += f"\n {line}"
        res += "\n " + " " * (longest_line + 2)
        return res
```

```
class SpreadsheetDocument(Document):
    def draw(self):
        res = ""
        res += ", ".join(str(cell).upper() for cell in self.data[0]) +␣
 ↪"\n-----------\n"
        for row in self.data[1:]:
            res += ", ".join(str(cell) for cell in row) + "\n"
        return res

haiku = TextDocument("haiku.txt", """Haikus are easy
But sometimes they don't make sense
Refrigerator""")

shopping = SpreadsheetDocument("shopping.ssd", [
    ("Name", "Price"),
    ("Bananans", 4.50),
    ("Bread", 1.95),
    ("Butter", 2.50),
])

documents = [haiku, shopping]

for doc in documents:
    print(doc.name)
    print(doc.draw())
```

```
haiku.txt

 Haikus are easy
 But sometimes they don't make sense
 Refrigerator

shopping.ssd
NAME, PRICE
-----------
Bananans, 4.5
Bread, 1.95
Butter, 2.5
```

3 Dealing with errors and exceptions

3.0.1 `try` / `except` / `finally` / `else`

No doubt you've already met *exceptions*, such as `IndexError` or `ValueError`.

[197]:
```
contacts = {"Alice": "+41001234567"}
#contacts["Bob"]  # Will raise a KeyError
```

However, there's nothing particularily ``bad'' about errors and exceptions. They are essentially just a secondary flow of information in parallel to regular function returns used when things don't go ``as they should normally''. A common place where exceptions will likely need to be handled is if you allow humans to enter data into your application. See this example:

[198]:
```
def beer_permitted():
    age = int(input(" > enter your age: "))    # a person could enter anything,␣
 ↪not just numbers
    return age >= 16
beer_permitted()
```

```
 > enter your age:  I don't wanna
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[198], line 4
      2     age = int(input(" > enter your age: "))    # a person could enter␣
 ↪anything, not just numbers
      3     return age >= 16
----> 4 beer_permitted()

Cell In[198], line 2, in beer_permitted()
      1 def beer_permitted():
----> 2     age = int(input(" > enter your age: "))    # a person could enter␣
 ↪anything, not just numbers
      3     return age >= 16

ValueError: invalid literal for int() with base 10: "I don't wanna"
```

This means that **exceptions are generally expected to happen**. For this reason, Python provides mechanisms to deal with them. You simply put code that might fail under certain conditions in a `try` block and then deal with a potential error in an `except` block:

[200]:
```
def beer_permitted():
    while True:
        try:
            age = int(input(" > enter your age: "))    # A ValueError could be␣
 ↪raised here...
            return age >= 16
        except ValueError:                             # ... and will be dealt␣
 ↪with here
            print("You must enter a number!")
beer_permitted()
```

```
 > enter your age:  I don't wanna!!!
```

```
You must enter a number!
```

```
 > enter your age:  No!
```

```
You must enter a number!
 > enter your age:  12
```

[200]: False

In the example above, we specifically say that we only catch `ValueError`s by saying `except ValueError`. However, you can also just use `except` as a blanket-statement. This is generally not recommended, because you want your code to crash if it's not used correctly, so that you can care for individual problems one at a time.

The general syntax for `try` / `except` / `finally` / `else` is:

```
try:
    # try executing this code that could fail with any kind of exception
except SpecificError:
    # execute this code if a SpecificError occured
except AnotherSpecificError:
    # execute this code if some OtherSpecificError occured
else:
    # execute this code only if no exceptions occured
finally:
    # always execute this code, whether an exception occured or not
```

Here's a concrete example illustrating all of this:

[201]:
```
def find_contact(contacts, name):
    return contacts[name]

def phone_as_int(number):
    return int(number[1:])

def call(name):
    print("Picking up phone...")
    contacts = { "Alice": "+49001234567", "Bob": "+41001234567", "Ivan":␣
 ↪"ivan@example.org"}
    try:
        phone = find_contact(contacts, name)
        number = phone_as_int(phone)
    except KeyError:
        print(f"{name} not in phonebook")
    except ValueError:
        print(f"{name}'s entry {phone} seems to be invalid")
    else:
        print(f"Calling {name}: {number}")
    finally:
        print("...Hanging up phone\n")

call("Alice")
call("Beatrice")
```

```
call("Ivan")
```

```
Picking up phone…
Calling Alice: 49001234567
…Hanging up phone

Picking up phone…
Beatrice not in phonebook
…Hanging up phone

Picking up phone…
Ivan's entry ivan@example.org seems to be invalid
…Hanging up phone
```

This is a very comprehensive example, but in real life, you primarily need to know just about `try` and `except SomeException`.

Using the `try`/`except` mechanism can also make your life easier. Consider this example:

You're retrieving data from some *unreliable* source, such as a human, and this data should be a number that can be converted to a hexadecimal number. As a reminder, we can convert any string representing a valid number of any base using the `int` function:

[202]:
```
print(int("101", 2))
print(int("10"))        # default is base 10
print(int("ff", 16))
```

```
5
10
255
```

Now consider how you would implement a function that will receive some arbitrary string and convert it without crashing, if it represents a valid hexadecimal number? Without using `try`/`except`, you would probably resort to checking whether all characters are valid (`all` is a function that takes a collection and checks if all values are *truthy*):

[203]:
```
def to_hex(string):
    if all(x in "0123456789abcdef" for x in string):
        return int(string, 16)
    else:
        return None
print(to_hex("af"))
print(to_hex("ag"))
```

```
175
None
```

But by using `try`/`except`, you can just go ahead and attempt the conversion without caring much about what is given to your function:

[204]:
```
def to_hex(string):
    try:
        return int(string, 16)
    except:
        return None
print(to_hex("af"))
print(to_hex("ag"))
```

```
175
None
```

3.0.2 Raising exceptions

Of course, when you write code, you may also want to `raise` your own exceptions and errors. Python provides a list of built-in exceptions here. As you can see, it's a hiearachy where many exceptions inherit from others. Given the rules of inheritance, that means that if you, for example `except LookupError`, this will catch both `KeyError` and `IndexError`, because they inherit from `LookupError`.

To cause an exception, you use the `raise` statement:

[205]:
```
raise Warning
```

```
---------------------------------------------------------------------------
Warning                                   Traceback (most recent call last)
Cell In[205], line 1
----> 1 raise Warning

Warning:
```

You can also add a custom message describing the problem in more detail:

[206]:
```
raise Warning("the user did a bad thing")
```

```
---------------------------------------------------------------------------
Warning                                   Traceback (most recent call last)
Cell In[206], line 1
----> 1 raise Warning("the user did a bad thing")

Warning: the user did a bad thing
```

Naturally, you can add you own exceptions, by simply inheriting from one of the existing Exception classes.

The following example is a bit redundant, but it illustrates the point:

[207]:
```
class ContactNotFoundError(LookupError):
    pass
```

```
def find_contact(contacts, name):
    if name not in contacts:
        raise ContactNotFoundError
    return contacts[name]

contacts = {"Alice": "+41001234567"}
try:
    find_contact(contacts, "Bob")
except LookupError:   # ContactNotFoundError inherits from LookupError, so it's␣
 ↪also caught here
    print("nope")
```

```
nope
```

Let's look at another example, this time involving classes:

[208]:
```
class Person:
    def __init__(self, name, age, job):
        if job is not None and age < 18:
            raise ResourceWarning("Child labor is illegal")
        self.name = name
        self.age = age
        self.job = job

    def __repr__(self):
        return f"{self.name} is {self.age} years old {f'and works as a {self.
 ↪job}' if self.job else ''}"

data_from_file = [
    ("Alice", 36, "Programmer"),
    ("Bob", 38, "Marketing Director"),
    ("Jimmy", 12, None),
    ("James", 18, "Waiter"),
    ("Lilly", 14, "Brain surgeon"),
]

population = []
invalid = []
for name, age, job in data_from_file:
    try:
        population.append(Person(name, age, job))
    except ResourceWarning:
        population.append(Person(name, age, None))
        invalid.append((name, age, job))
print(population)
print(invalid)
```

```
[Alice is 36 years old and works as a Programmer, Bob is 38 years old and works
```

```
as a Marketing Director, Jimmy is 12 years old , James is 18 years old and works
as a Waiter, Lilly is 14 years old ]
[('Lilly', 14, 'Brain surgeon')]
```

4 More on functions

4.0.1 Functions can take ``optional'' arguments

There are two ways function arguments are specified in Python. What you've come to know and love are *positional* arguments (like `x` and `n` in the following example):

[219]:
```
def power(x, n):
    return x**n
print(power(3,2))
```

```
9
```

But Python also supports what it calls *keyword* arguments:

[220]:
```
def power(x, n=2):
    return x**n
print(power(3))      # n is not passed, the default (2) is used
print(power(3, 3))   # n is passed, it is used instead of the default
```

```
9
27
```

For *keyword* arguments, you specify a default value as part of the function signature. If the user does not pass a value for that argument, the default will be used.

Note that keyword arguments must always come after positional arguments.

Here's another example:

[221]:
```
def format(name, age, job=None, bold=False):
    res = f"""{name} is {age} years old {f'and works as a {job}' if job else␣
 ↪''}"""
    if bold:
        print(res.upper())
    else:
        print(res)

format("Alice", 31)
format("Bob", 29, "Engineer")
format("Cooper", 55, "Banker", True)
```

```
Alice is 31 years old
Bob is 29 years old and works as a Engineer
COOPER IS 55 YEARS OLD AND WORKS AS A BANKER
```

Watch out! A tricky situation will arise if you use a *mutable* value as a default parameter! The reason is that whatever you put into your signature as a default parameter is only instantiated *once*

when the function signature is passed.

Say you want some parameter to be an empty list by default:

[222]:
```
class Student:
    def __init__(self, name, subjects=[]): # this seems intuitive...
        self.name = name
        self.subjects = subjects
    def enroll(self, subject):
        self.subjects.append(subject)

bob = Student("Bob")     # not passing a value for 'subjects' so the default␣
 ↪([]) is used
print(bob.subjects)
bob.enroll("Info1")
print(bob.subjects)
```

```
[]
['Info1']
```

This appears to work fine, but watch what happens when we now create another instance of `Stundent`:

[223]:
```
alice = Student("Alice")    # again, not passing a value for subjects
print(alice.subjects)
alice.enroll("Bio3")
print(bob.subjects)
```

```
['Info1']
['Info1', 'Bio3']
```

`alice` and `bob` appear to share the same list for `subjects`! This is because the method signature

`def __init__(self, name, subjects=[]):`

is only executed once, when Python interprets the class definition. This sets `subjects` to a new empty list, and any `Student` instance created using this constructor will refer to **the same** list object.

So it's important that you only use *immutable* values for defaults. If you want to have an empty list by default, you would need to work around this, for example using `None`:

[224]:
```
class Student:
    def __init__(self, name, subjects=None): # this is the correct way!
        if subjects is None:
            subjects = []      # this creates a new list every time __init__ is␣
 ↪called
        self.name = name
        self.subjects = subjects
    def enroll(self, subject):
        self.subjects.append(subject)
```

```
bob = Student("Bob")  # subjects will be set by calling list(), which makes a␣
 ↪new list (every time!)
print(bob.subjects)
bob.enroll("Info1")
print(bob.subjects)
alice = Student("Alice")
print(alice.subjects)
```

```
[]
['Info1']
[]
```

4.0.2 You can pass arguments out of order if you mention their names explicitely

You can explicitely mention argument names when *calling* a function. This allows you to specify arguments out of order. This is particularily useful when using functions that take many, many parameters, and maintaining an order is inconvenient:

[225]:
```
def vital_signs(name, age, heart_rate, blood_pressure, o2_level,␣
 ↪sleeping=False):
    ok = True
    if heart_rate < 60 or heart_rate > 100:
        ok = False
    if blood_pressure < 60 or blood_pressure > 140:
        ok = False
    if o2_level < 95:
        ok = False
    return f"{name} ({age}) years old is {'in danger' if not ok else 'ok'}"
print(vital_signs("Bob", 23, 150, 200, 99, True))
print(vital_signs("Ann", 23, sleeping = False, o2_level = 98, blood_pressure =␣
 ↪90, heart_rate = 80))
```

```
Bob (23) years old is in danger
Ann (23) years old is ok
```

From here to the bottom of this script follows content that will not be tested for in the exam

4.0.3 You can *get* all positional and keyword arguments at once

Consider the following function:

[226]:
```
def team(name1, name2, name3, color, penalty=0):
    members = [name1, name2, name3]
    print(f"{color.capitalize()} team has penalty {penalty} and these members:␣
 ↪{name1}, {name2}, {name3}")
team("Bob", "Alice", "Dean", "red", 5)
```

```
Red team has penalty 5 and these members: Bob, Alice, Dean
```

Instead of spelling out each positional and keyword argument, you can use the * and ** notation, as follows:

[227]:
```
def team(*args, **kwargs):
    members = args[0:3]
    print(f"{args[3].capitalize()} team has penalty {kwargs['penalty']} and␣
 ↪these members: {', '.join(members)}")
team("Bob", "Alice", "Dean", "red", penalty=5)
```

```
Red team has penalty 5 and these members: Bob, Alice, Dean
```

Basically, `*args` is a list containing all positional arguments, which you can retrieve by index (e.g., `args[3]`) and `**kwargs` is a dictionary containing all keyword arguments, which you can retrieve as usual by key access (e.g., `kwargs['penalty']`.

4.0.4 You can *set* all positional and keyword arguments at once

The same, just reversed, is true when calling a function. You can pass in a list or dict respectively to set all positional and/or keyword arguments:

[228]:
```
def team(name1, name2, name3, color, penalty=0):
    members = [name1, name2, name3]
    print(f"{color.capitalize()} team has penalty {penalty} and these members:␣
 ↪{name1}, {name2}, {name3}")
my_args = ["Bob", "Alice", "Dean", "red"]
my_kwargs = {'penalty': 5}
team(*my_args, **my_kwargs)
```

```
Red team has penalty 5 and these members: Bob, Alice, Dean
```

5 Multiple inheritance

Some programming languages, including Python, allow inheriting from more than one class at the same time. In Python, the precedence rule for multiple inheritance are fairly simple: if multiple super-classes specify the same method or attribute, it's always the *left-most* one that's inherited:

[25]:
```
class A:
    def info(self):
        print("info A")
    def method_a(self):
        print("only in A")

class B:
    def info(self):
        print("info B")
    def method_b(self):
        print("only in B")

class Z(A, B):
```

```
    pass        # Z inherits everything from A and B but adds nothing by itself

z = Z()
z.method_a()
z.method_b()
z.info()        # inherited from A, because it's left-most in Z(A, B)
```

```
only in A
only in B
info A
```

However, remember that `self` is always just some concrete instance, i.e., some object, and that `z.info()` is just syntactic sugar for `Z.info(z)`. Both of these pass the `z` instance into `Z.info` as the `self` parameter.

This means that you can still execute `B`'s `info` method for a `Z` object, by explicit invocation:

[26]:
```
B.info(z)
```

```
info B
```

This becomes particularly relevant, when using constructors in multiple inheritance. Here we have two classes `A` and `B`, each with their own specific constructor:

[37]:
```
class A:
    def __init__(self, int1, int2):
        self.int1 = int1
        self.int2 = int2
    def info(self):
        return f"A info: {self.int1}, {self.int2}"

class B:
    def __init__(self, str1, str2):
        self.str1 = str1
        self.str2 = str2
    def info(self):
        return f"B info: {self.str1}, {self.str2}"

a = A(10, 20)
b = B("beep", "boop")
print(a.info())
print(b.info())
```

```
A info: 10, 20
B info: beep, boop
```

If we now inherit from both `A` and `B`, then `super().__init__` will be the left-most inherited constructor, which is `A.__init__`.

[38]:
```
class Z(A, B):
    def __init__(self, int1, int2, str1, str2):
        super().__init__(int1, int2)     # left-most (A's) constructor
z = Z(88, 99, "zeep", "zoop")            # the two strings are not stored!
print(z.int1)
#print(z.str1)
```

```
88
```

So when using multiple inheritance, you're better off just calling the super constructor explicitely by the class name, rather than using `super()`:

[41]:
```
class Z(A, B):
    def __init__(self, int1, int2, str1, str2):
        A.__init__(self, int1, int2)
        B.__init__(self, str1, str2)
    def info(self):
        return f"{A.info(self)}, {B.info(self)}"
z = Z(88, 99, "zeep", "zoop")
print(z.int1)
print(z.str1)
z.info()
```

```
88
zeep
```

[41]:
```
'A info: 88, 99, B info: zeep, zoop'
```

Here's a more meaningful example for how multiple inheritance might be used. Imagine you have `Time` and `Date` implementations that you can use independently:

[55]:
```
class Time:
    def __init__(self, h, m, s):
        self.hour = h
        self.minute = m
        self.second = s

    def add_hour(self):
        self.hour += 1

    def add_minute(self):
        self.minute += 1
        if self.minute == 60:
            self.add_hour()
            self.minute = 0

    def add_second(self):
        self.second += 1
        if self.second == 60:
```

```
            self.add_minute()
            self.second = 0

    def __str__(self):
        return f"{self.hour:02}:{self.minute:02}:{self.second:02}"

t = Time(19, 59, 59)
print(t)
t.add_second()
print(t)
```

```
19:59:59
20:00:00
```

[56]:
```
class Date:
    def __init__(self, y, m, d):
        self.year = y
        self.month = m
        self.day = d

    def add_year(self):
        self.year += 1

    def add_month(self):
        self.month += 1
        if self.month == 13:
            self.add_year()
            self.month = 0

    def add_day(self):
        self.day += 1
        # left as an exercise to the reader

    def __str__(self):
        return f"{self.year:04}-{self.month:02}-{self.day:02}"

d = Date(1999, 12, 31)
print(d)
```

```
1999-12-31
```

And now you want to create a `DateTime` class. Since you already implemented almost everything, you can inherit from both `Date` and `Time`:

[61]:
```
class DateTime(Date, Time):
    def __init__(self, year, month, day, hour, minute, second):
        Date.__init__(self, year, month, day)
        Time.__init__(self, hour, minute, second)
```

```
    def __str__(self):
        return f"{Date.__str__(self)} {Time.__str__(self)}"

    def add_hour(self):        # overriding Time.add_hour!
        Time.add_hour(self)
        if self.hour == 24:
            self.add_day()
            self.hour = 0

dt = DateTime(1999, 1, 1, 23, 59, 59)
print(dt)
dt.add_second()
print(dt)
```

```
1999-01-01 23:59:59
1999-01-02 00:00:00
```