COMP 1510
Winter 2022

Week 12: Decorators, closures, ducks, and exceptions

Agenda for week 12

1. Function decorators, inner functions, and closures
2. Compiling vs interpreting
3. Duck typing (static vs dynamic) and strong vs weak typing
4. sys.args for command line arguments
5. Passing command line arguments to the main function
6. Exceptions
    a) try-except-else-finally
    b) Unit testing: testing for expected exceptions
    c) Exception hierarchy and commonly used exceptions
7. Guard clauses are wasteful (LBYL vs EAFP)
8. Modules and packages
9. Refactoring
10. Code smells and the refactoring catalog:
    a) The basics
    b) Encapsulation
    c) moving features around
    d) Organizing data
    e) Clarifying logic
    f) Refactoring simple APIs

Tuesday, March 22, 2022

FUNCTION DECORATORS, INNER FUNCTIONS, AND CLOSURES

Let's start with functions

- In Python functions sometimes return values
- The return value is sometimes but not always determined by the arguments the function receives
- If we don't explicitly return something from a function, it will automatically return anyway, and implicitly *return None*
- Python functions can also produce side effects
  - print( ) is a good example: it returns None (bet you didn't know that!), and before it does, it outputs something to the console
  - A side effect is a permanent change made by the function
  - It's something we should include as a postcondition

Recall that functions are first-class objects

- In Python we can treat functions as objects
- Each function is stored in memory, and it has an address
- We can use the address as a reference to the function
- We can pass functions around like arguments!

Example

```
def say_hello(name):
    return f"Hello {name}"

def say_bye(name):
    return f"See you later {name}"

def address_students(address_func):
    return address_func(”Students!")
```

How would we invoke address_students?

```
>>> address_students(say_hello)
'Hello Students!'

>>> address_students(say_bye)
‘See you later Students!'
```

Introducing inner (nested) functions

- In Python we can define functions **inside** other functions
- We call these inner functions
- We might choose to use inner functions because:
  1. We want to encapsulate them and hide them from the global scope, i.e., we don't want them available to other functions
  2. Inner functions can be used as closures. Closures are rampant in programming. The inner function remembers the state of its environment in which it is called.*

* Let's examine closure.py AFTER the next two slides

An example of an inner function (actually 2!)

```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
```

Execution produces this:

```
>>> parent()
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

- Q: What happens if we change the sequence of the inner functions inside the parent( ) function?
- Q: What happens if we try to invoke either of the #_child( ) functions from outside the parent( ) function

Remember we can return functions too!

- We can return a function from a function
- Let me say that again:

We can use functions as return values!

- This can be a closure (let's look at closure.py now)
- Or it can be a function that does not retain any information about its environment
- For example, we can evaluate a parameter and select a different return function based on what it is… (see next slide)

Example

```
def english_or_korean(language):
    def english_version():
        return "I really like gochujang"

    def korean_version():
        return "나는 고추장이 좋아"

    if language == "한국어":
        return korean_version
    else:
        return english_version
```

Here is some sample output:

```
>>> first = english_or_korean("English")
>>> second = english_or_korean("한국어")
>>> first
<function english_or_korean.<locals>.english_version at
0xabc>
>>> second
<function english_or_korean.<locals>.korean_version at 0x123>
>>> first()
'I really like gochujang'
>>> second()
'나는 고추장이 좋아'
```

Now we can look at decorators

- You now understand that the Python function is just like any other object in Python
- It's time for your first

decorator

- It's going to feel like a little pinch

Decorator number 1

```
def my_decorator(some_other_function_I_am_decorating):
    def wrapper():
        print("Something is happening before my function is called.")
        some_other_function_I_am_decorating()
        print("Something is happening after my function is called.")
    return wrapper

def say_cake():
    print("Cake! Gâteau! 케이크!, 蛋糕! bánh kem! Торт! केक! کیک 🎂")

say_cake = my_decorator(say_cake) # This is where the magic happens
```

And this is how it works:

```
>>> say_cake()
Something is happening before the function is called.
Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂
Something is happening after the function is called.

>>> say_cake
<function my_decorator.<locals>.wrapper at 0x257>
```

What happened here?

- The actual decoration takes place at this line:

```
say_cake = my_decorator(say_cake)
```

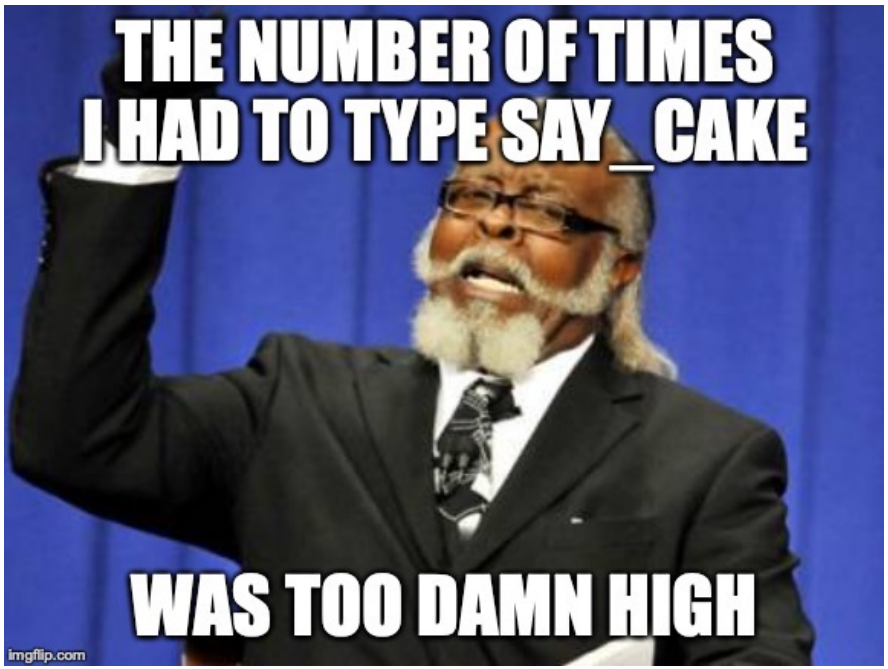
- The identifier say_cake now stores the address of the wrapper( ) function instead of the original say_cake function
- The wrapper( ) function was returned from my_decorator( )
- And wrapper( ) maintains a reference to the original say_cake( ) function
- ***The decorator encapsulates a function and enhances its behaviour!***

Decorator number 2

```
from datetime import datetime

def not_during_the_night(function_to_execute):
    def wrapper():
        if 7 <= datetime.now().hour <= 22:
            function_to_execute()
        else:
            print(“No raiding the fridge at night, Chris!”)
    return wrapper

def say_cake():
    print("Cake! Gâteau! 케이크!, 蛋糕! bánh kem! Торт! केक! کیک 🎂 ")

say_cake = not_during_the_night(say_cake)
```

Motivating annotations (pie syntax)

- The way we decorated say_cake( ) was a bit clunky
- We had to type say_cake too many times
- And the decoration can be hard to find in all that code



Finally: annotations!

Look familiar?
This is an annotation
It uses "*pie syntax*"

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_cake():
    print("Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂 ")
```

@my_decorator is just a shortcut for say_cake = my_decorator(say_cake)

Decorate a function with different decorators

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice

@do_twice
def say_cake():
    print("Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂 ")

>>> say_cake()
Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂
Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂
```

What if our function accepts parameters?

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice

@do_twice
def greet_with_cake(name):
    print(f"Cake for {name}!")

>>> greet_with_cake(”everyone”)
```

What gets printed?

What if our function accepts parameters?

```
def do_twice(func):
    def wrapper_do_twice():
        func()
```

Nothing! Oh noooo!

```
>>> greet_with_cake("everyone")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

```
@do_twice
def greet_with_cake(name):
    print(f"Cake for {name}!")

>>> greet_with_cake(”everyone”)
```

The issue

- The inner function wrapper_do_twice( ) doesn't accept any parameters
- But name="everyone" must be passed to the nested function it has wrapped
- Q: How do we pass the parameter(s)?
- A: We use:
  1. *args (called star-args)
  2. **kwargs (called K. W. args)

Star-args and K. W. args

- *args is used to send a non-keyworded variable length argument list to a function
- **kwargs allows you to pass a variable number of keyworded arguments to a function
- We should use **kwargs if e want to handle named arguments in a function
- I think we should use both!

The fix slide I of II

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice

@do_twice
def say_cake():
    print("Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂 ")
```

The fix slide II of II

```
@do_twice
def greet_with_cake(name):
    print(f"Cake for {name}!")

>>> say_cake()
Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂
Cake! Gâteau! 케이크!, 蛋糕! bánh ngọt! Торт! केक! کیک 🎂

>>> greet_with_cake("everyone")
Cake for everyone!
Cake for everyone!
```

What if our function returns a value?

- Right now, the inner function wrapper( ) doesn't return a value
- But the function it wraps around does return a value!
- **Q**: How do we return a value?
- **A**: Use a return statement and invoke our wrapped function with our friends:
  1. *args (remember we say star-args)
  2. **kwargs (remember we say K. W. args)

The fix slide I of II

```
def decorate_and_return(func):
    def wrapper(*args, **kwargs):
        print("Something is happening!")
        return func (*args, **kwargs)
```

The fix slide II of II

```
@decorate_and_return
def return_greeting(name):
    return f"Hi {name}"

>>> hi_justin = return_greeting("Justin")
>>> print(hi_justin)
Something is happening!
Hi Justin
```

Practical application: timing something!

```
import time


def timer(func):
    """Print the runtime of the decorated function"""
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__} in {run_time:.4f} secs")
        return value
    return wrapper_timer
```

How would I use this? I'd decorate my function!

```
@timer
def waste_some_time(num_times):
    for value in range(num_times):
        sum([value **2 for value in range(num_times)])

>>> waste_some_time(1)
Finished 'waste_some_time' in 0.0010 secs
>>> waste_some_time(999)
Finished 'waste_some_time' in 0.3260 secs
```


Decorators remind me of that @patch stuff with unit tests, Chris…

@patch

- That's right! We've been decorating our unit tests with @patch
- We've been enclosing our unit tests inside a wrapper function
- The wrapper function (patch) was written to help our tests
- We do this to try to minimize"
    1. the number of dependencies
    2. aka the degrees of freedom
    3. aka the number of moving parts
    4. and the amount of boilerplate code.
- We do this by "patching in" a prepared (mock) value when the function being tested calls another function!

@patch: **kwargs and return_value

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish’

>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

@patch: **kwargs and side_effect

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)

>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

* https://docs.python.org/3/library/unittest.mock.html

COMPILING VS INTERPRETING

Scripting languages

- Until the 1960s, languages were ***compiled*** (do you remember who wrote the first compiler?)
- During the 1960s and 1970s, programmers began creating ***scripting languages*** to execute programs without compiling them first
- A ***script*** is a program whose instructions are executed by another program called an ***interpreter***
- Interpreted execution can be a bit slow but has advantages:
    1. Compilation can be very slow
    2. We can execute the same script on different processors if each processor has an interpreter installed.

Some Python history

- In the late 1980s, ***Guido van Rossum*** began creating a scripting language and interpreter called Python
- Guido derived Python from an existing language called ***ABC***
- The name Python came from the TV show ***Monty Python***.
- The goals for the language included:
  1. Simplicity
  2. Readability
  3. As much power and flexibility as other scripting languages like Perl that were popular at that time.

(Aside: some more Python history)

- Python ***1.0*** was released in ***1994*** with support for some functional programming constructs derived from Lisp
- Python ***2.0*** was released in ***2000*** and introduced automatic memory management (garbage collection), and features from Haskell and other languages
- Python ***3.0*** was released in ***2008*** to rectify various language design issues
- Python 3.0 is not backwards compatible, it was a HUGE change!
- Python is an open-source language!

Python is special – it uses dynamic typing

- A Python programmer can use any type of object as an argument to a function
- Consider a function add(x, y) that adds the two parameters

```
def add(a, b):
    return a + b
```

- We can invoke this function using ints or using strings or lists!

```
print(add(1, 1))
print(add('nanoo', 'nanoo'))
```

More dynamic typing

- Python uses *dynamic typing* to determine the type of objects as a program executes
- For example, the consecutive statements *num = 5* and *num = '7'* first assign the address of an int to num, and then the address of a string!
- The type of object num is bound to can change, depending on the value it references (the value at the address it stores)
- The interpreter is responsible for checking that all operations are valid as the program executes
- If the function call add(5, '100') is evaluated, the interpreter generates an error when we try to add the string to an integer.

Static typing

- In contrast to dynamic typing, many other language like C, C++, and Java use *static typing*
- Static typing requires the programmer to define the type of every variable and every function parameter in a program's source code
- For example *int answerToEverything = 42* to declare an int variable
- When the source code is compiled, the compiler attempts to detect operations that are not type-safe, and halts the compilation process if such an operation is found
- We are not allowed to execute the program. No executable code is produced until the compiler errors are eliminated.

Which is better?

- *Dynamic typing* typically gives the programmer more flexibility than static typing
- *Dynamic typing* can potentially introduce more bugs, because there is no compilation process that checks types
- If all we have is a hammer, everything looks like a nail
- *We need lots of tools*
- Python is just one (very super excellent amazing) tool
- You will have many tools in a few months/years!

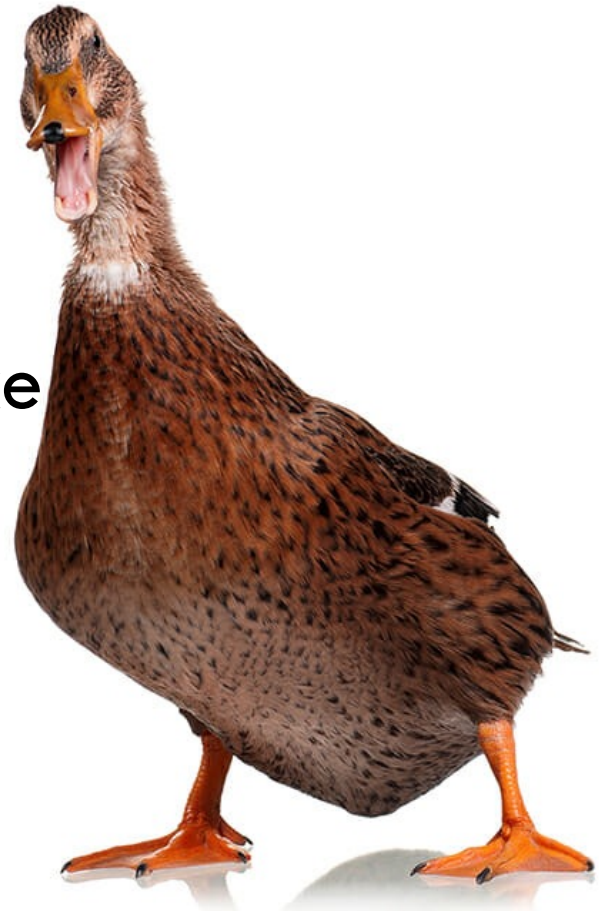
DUCK TYPING

Duck typing

Some people say that Python uses ***DUCK TYPING***:

"If a bird walks, swims, and quacks like a duck, then call it a duck"

For example, if an object can be concatenated, indexed, and converted to Unicode, that is, if it does everything a string can do, then let's treat the object like a string.



COMMAND LINE ARGUMENTS

The sys module

- Python has a library
- It is a library of modules instead of books
- These modules contain helpful code
- Programmers love re-using code
- The sys module contains a list called sys.argv
- We can pass information to a module (our program) on the command line!
- Let's look at sysdemo.py together

What about command line arguments

- Remember that every module (file) must have a main function
- We can use the main function to test the code inside the file:
  - Demonstrate that it works
  - Illustrate how to use the functions
- But how do we pass command line arguments to the main function when we execute a program?
- We use a little algorithm:
  1. Pass the command line argument(s) as parameters to main
  2. Main accepts the parameters and passes them to the variables or functions that need them.

Passing command line arguments to main:

sample.py

```
import sys

def main(number_of_players, difficulty_level):
    team = create_players(number_of_players)
    environment = build_game(difficulty_level)
    play(team, environment)

if __name__ == “__main__”:
    main(sys.argv[1], sys.argv[2])
```

EXCEPTIONS

We've seen some exceptions already

- Python uses special objects called exceptions to manage errors that arise during program execution
- If we don't do something about the exception, the program crashes and displays a **traceback**
- The traceback reports the name of the exception that took place
- We can deal with exceptions by using a new control structure called the try-except block

Sample try-except block

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

That's it?

Yep. That's it.

We do this to prevent (control!) crashes

```
print(”Enter two numbers, and I'll divide the first by the second.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        return
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number)
        print(answer)
    except ZeroDivisionError:
        print("You can't divide by 0!")
```

Don't let users see tracebacks

- *Tracebacks are like the Windows blue screen of death*
- Non-technical users are confused by tracebacks
- Crashes strike terror in the hearts of users
- Crashes undermine confidence in your software

- Test your code
- Anticipate what can go wrong
- Plan your recovery

The else block

Any code that depends on the try block executing correctly should go in the else block:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        return
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    except ValueError:
        print('You must give me two integers!')
    else:
        print(answer)
```

So what goes in the try block?

- *Only code that might cause an exception to be raised*
- Don't put EVERYTHING in the try, just one line of code
- Everything else goes before the try or in the **else** block
- (Remember the except block tells us how to manage a certain exception)
- There's something more, too. There's a finally block!

Finally, the finally!

- We can include a finally block!
- If a finally clause is present, the finally clause will execute as the last task before the try statement completes
- The finally clause always runs, even if the try statement produces an exception

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("This always gets printed")
```

What about failing silently?

- What if we don't need to report every exception?
- Some developers will *catch the exception and do nothing*
- We call this failing silently:
  1. Use a try-except statement
  2. Inside the except block, do nothing
- *Don't do this. The user gets no feedback!*

When do we report errors?

- Well-written, properly tested code is not very prone to internal error such as syntax or logical errors
- But anytime we interact with a user, we introduce the possibility of run-time errors
- Giving users information they can't use will decrease the usability of your program
- **Recover** gracefully, **warn** the user if they did something they shouldn't, and tell them how to **avoid** committing the same error again.

Built-in exceptions

- Check this out:

  https://docs.python.org/3/library/exceptions.html#built-in-exceptions

- Familiarize yourself with the exceptions here
- How many have you seen?
- How many make no sense to you?
- What is the difference between an exception and a warning?

UNIT TESTING: EXCEPTIONS

How to test a function that raises an exception

- What if a function is supposed to raise an exception in response to specific input
- This is something we need to prove, i.e., we need to write a unit test
- Which assertion should we use?

```
with self.assertRaises(SomeException):
    do_something()
```

1a. Asserting that an expected error occurs

```
def test_is_positive_str_typeerror(self):
    with self.assertRaises(TypeError):
        functions.is_positive('not an integer!')
```

1. Suppose we are testing an is_positive function that accepts an int
2. We instructed the developer who coded up is_positive to raise a TypeError if a user passes it something that is not an int
3. We will write a test that will only pass if the correct type of error is raised by the interpreter during execution
4. Note in this example the use of the with keyword, and the assertRaises assertion method
5. We want to assert that invoking the function with that specific input raises that specific error.

1b. Asserting that an expected error occurs

```
def test_factorial_valueerror(self):
    with self.assertRaises(ValueError):
        my_factorial(-2)
```

1. This is the same sort of unit test, but this time we are trying to calculate the factorial of a negative integer
2. The math module's factorial function raises a ValueError when we use it to do this
3. This unit test asserts that when we try to pass a -2 to our my_factorial function, it will raise the ValueError that it received math module's factorial function
4. If our my_factorial function does NOT raise a ValueError, the unit test will fail.

A LITTLE PHILOSOPHY

LBYL vs EAFP

- Many languages check data types
- For example, a function that accepts a list will check that the address of a list was actually passed to it as an argument
- If the argument is not a list, the runtime interpreter generates an error
- We call this ***Look Before You Leap*** (LBYL)
- That's a choice
- Another choice is the avoid doing this
- We call this ***Easier to Ask For Forgiveness than Permission***
- Python encourages an EAFP culture

Easier to Ask For Forgiveness than Permission

- EAFP means that you should just start by doing what you expect to work
- If an exception might be thrown from the operation, then catch the exception and deal with that fact
- We call this idiomatic Python. It's… Pythonic.
- An idiom is a characteristic way of doing things
  - Pythonistas use idiomatic Python
  - C developers use C idioms
  - Java developers use Java idioms…

Compare: accessing a dictionary value

LBYL: ❌

```
if "key" in dict_:
    value += dict_["key"]
do_something(value)
```

EAFP: ✅

```
try:
    value += dict_["key"]
except KeyError:
    print("No key!")
else:
    do_something(value)
```

Don't check. Assume. Handle the error if the assumption is wrong. And carry on.

Compare: accessing a dictionary value

LBYL: ❌

```
if file_exists:
    process_file()
```

EAFP: ✅

```
try:
    process_file()
except FileNotFoundException:
    // react appropriately
```

Don't check. Assume. Handle the error if the assumption is wrong. And carry on.

MODULES AND PACKAGES

Module (review)

- A python **source file** is called a module
- A module is a collection of **functions, declarations, and statements** grouped together in a file
- To use a module, we must import it
- After a module has been imported, we refer to its contents using the following syntax:

```
module_name.function_name(parameters)
module_name.CONSTANT
```

Importing modules

- There's more than one way
- Suppose we want to import the math module or some of its contents. We can use the following:

```
1. import math
2. from math import pi
3. import math as mathematics
4. from math import pow as power, sin as sinus
5. from math import *
```

- *(But don't use the wildcard (*) because it is considered lazy, sloppy programming and I will dock marks from your grades)*

Module design

- Put functions and variables that logically belong together in the same module
- If there isn't a logical connection, don't put them in the same module
- Our goal is to *minimize dependencies*
- Rules of thumb (more like heuristics):
    1. If a module has less than a handful of things in it, it's probably too small
    2. If you can't sum up the contents and purpose of a module in a one- or two-sentence docstring, it's probably too large.

Module design

- Modules usually only contain constants, functions, and possibly some definitions
- Modules can contain initialization statements too
- Any initialization statements are executed when we import the module
- At the beginning of the term, we observed that this can be troublesome and it motivated the main method
- But sometimes we do this intentionally (suppose we are setting a constant to be the current time, for example…)

Contents of a module

We can find out what a module contains by using the dir( ) function:

1. Try this:

```
>>> import math
>>> dir(math)
```

2. Try this too (what is different?):

```
>>> import math
>>> dir()
```

3. And what about this:

```
>>> import builtins
>>> dir(builtins)
```

Chris, what's a package?

- I'm very glad you asked
- Suppose I create a MUD
- I've got dozens and dozens of modules
- Keeping them in one folder is not a good idea
  - Difficult to maintain
  - Difficult to share
  - Not very 'modular'
- *We like to organize our modules into packages.*

Chris, what's a package?

- A package is a named source code directory that contains:
  1. One or more more Python source files (modules)
  2. An additional Python file named __init__.py
- We use packages to create a "structured namespace"
- *A namespace is a "set of symbols used to organize objects"*
- We are already using namespaces: modules!
- We can nest modules in packages in hierarchies
- That means we can nest namespaces too!

An example design:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

What's that __init__.py file?

- A convention
- Required so Python treats the directory as a *package*
- Usually empty, but provides a place for us to declare variables and create objects that can be used by all the modules in the package
- This is a good place to define constants that are used by the entire program, for example
- Rather than re-defining the same constants in each module, define the constant once in __init__.py!

README: https://docs.python.org/3/tutorial/modules.html#packages

What else can we do with packages?

- A package's `__init__.py` code can define a list named `__all__`
- If there is a list called `__all__` inside the `__init__.py` file, the interpreter treats it as a list of module names that should be imported when `from package import *` is used (yuck!)
- If `__all__` is not defined, the statement `from package import *` does not import all submodules from the package into the current namespace (tricky and insidious error if you don't know about it!)
- The interpreter will only ensure that the package has been imported (possibly running any initialization code in `__init__.py`), and then imports whatever names are defined in the package.

An example design:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Maybe we will use this in sound/effects/__init__.py:

```
__all__ = ["echo", "surround", "reverse"]
```

Importing nested packages

- Suppose we have this package structure:

```
directory/my_package/__init__.py
directory/my_package/my_module.py
```

- We can import my_module like this:

```
import my_package.my_module
from my_package import my_module
```

README: https://docs.python.org/3/tutorial/modules.html#packages

Why use packages?

- Structure
- Organization
- Maintainability
- Independent namespaces
- "Dotted module names"
- Best practice: use `from package import specific_submodule` unless the importing module needs to use submodules with the same name from different packages.

REFACTORING

Refactoring

- ***Improving the design*** of existing code
- The process of restructuring existing computer code, i.e., changing the factoring, *without changing its observable behaviour*
- Intended to *improve non-functional attributes* of the software:
  - Makes software easier to understand
  - Prevents the design from decaying
  - Helps us find bugs.

How do we refactor our code?

- *We apply a series of refactorings, i.e., we change a function declaration or e remove dead code, etc., etc…*
- *There is an "official catalogue" of refactorings that developers use*
- You might spend a couple of hours refactoring a small project
- You would probably apply several dozen refactorings during this time
- You should try to code by continuously:
  - adding and testing functionality
  - refactoring once in a while.

How often is “once in a while”?

Observe the *Rule of Three*:

1. The **first time** we do something, we just do it
2. The **second time** we do something similar we wince at the duplication, but we do the duplicate thing anyway
3. The **third time** we do something similar, we refactor.

CODE SMELLS

Refactoring eliminates “codes smells”

https://sourcemaking.com/refactoring/smells

Some code smells (there are over 20) include:

1. Mysterious names
2. Duplicated code
3. Long functions
4. Long parameter lists
5. Global data (just shoot me now)
6. Mutable data

THE "BASICS"

Where's the mighty list of refactorings, Chris?

BEHOLD: https://refactoring.com/catalog/

We call it a catalog, Karen.

A catalog of refactorings.

There are 65+ refactorings in the catalog.

I'll introduce you to a few. You must read the rest. **MANDATORY READING!**

Each refactoring is applied to solve a problem.

There is a simple algorithm for applying each refactoring correctly.

1. Extract a function

1. Create a new function, and give it a meaningful identifier
2. Copy the extracted code from the source function into the new target function
3. Scan the extracted code for references to any variables that are local in scope to the source function that will not be in scope for the extracted function, and pass them as parameters
4. Replace the extracted code in the source function with a call to the target function – remember to store the return value if there is one
5. Compile and test
6. https://sourcemaking.com/refactoring/extract-method

2. Inline a function

- Sometimes, we do come across a short function that is so short it shouldn't be a separate function
- Or, we refactor the body of the code into something that is just as clear as the name
- Indirection can be helpful, but needless indirection is irritating, so we can:
    1. Find all the callers of the function
    2. Replace each call with the function's body
    3. Test after each replacement
    4. Remove the function definition.

https://sourcemaking.com/refactoring/inline-method

3a. Change a function declaration

- If we see a function with the wrong name, it is imperative that we change it as soon as we choose a better identifier
- Or we may want to change the parameter list for the function
- We like to minimize:
  - the amount of coupling between functions
  - how much each module needs to know each each other
  - the number of parameters being passed between functions.

https://refactoring.com/catalog/changeFunctionDeclaration.html

3b. How to change a function declaration

Simple approach:

1. If you're removing a parameter, ensure it isn't referenced in the body of the function
2. Change the method declaration to the desired declaration
3. Find all references to the old method declaration, update them to the new one
4. Test.

3c. How to change a function declaration

Migration approach:

1. If necessary, refactor the body of the function to make it easy to do the following extraction step
2. Use Extract Function on the function body to create a new function
3. If the new function will have the same name as the old one, give the new function a temporary name that's easy to search for
4. If the extracted function needs additional parameters, use the simple mechanics to add them
5. Test
6. Apply Inline Function to the old function
7. If you used a temporary name, use Change Function Declaration again to restore it to the original name
8. Test.

MOVING FEATURES AROUND

1a. Move a function

- We should move a function when it references elements in other contexts more than the one it currently resides in
- Or we may move a function because of where its callers live, or where we need to call it from
- A function defined as a helper inside another function may have value on its own, so it may be worth moving it to somewhere more accessible, i.e., into the enclosing scope
- Examine the current and candidate contexts for that function
- *Although it can be difficult to decide where the best place for a function is, the more difficult this choice, often the less it matters*
- https://sourcemaking.com/refactoring/move-method

1b. How to move a function

1. Examine all the program elements used by the chosen function in its current context and consider whether they should move too.
2. Copy the function to the target context and adjust it to fit in its new home
3. Figure out how to reference the target function from the source context
4. Turn the source function into a delegating function
5. Test.

2a. Move statements into a function

- Removing duplication is one of the best rules when writing healthy code
- When we see the same code executed every time we call a particular function, we should combine that code with the function itself
- We move statements into a function when we can best understand these statements as part of the function
- Any future modifications to the repeating code can be done in one place and used by all the callers
- https://refactoring.com/catalog/moveStatementsIntoFunction.html

2b. How to move statements into a function

1. If the target function is only called by the source function, just cut the code from the source, paste it into the target, test, and ignore the rest of these mechanics
2. If you have more callers, use Extract Function on one of the call sites to extract both the call to the target function and the statements you wish to move into it -- give it a name that's transient
3. Convert every other call to use the new function
4. Test after each conversion
5. When all the original calls use the new function, use Inline Function to inline the original function completely into the new function, removing the original function
6. Rename Function to change the name of the new function to the same name as the original function.

3. Remove dead code

- Unused code is a burden
- We call unused code dead code
- There's no good reason to keep it because we can revisit it using version control
- It generates cognitive debt
- ***Remove it!***
- https://refactoring.com/catalog/removeDeadCode.html

SIMPLIFYING AND CLARIFYING

1. Split a variable

- Variables are often assigned new values
  - Loop variables change during each iteration
  - Collecting variables store a value, i.e., a sum, during execution of a function
- Sometimes a variable is used to store completely different things
- ***Don't ever do this!***
- It's too confusing
- We split the variable instead!
- https://sourcemaking.com/refactoring/split-temporary-variable

2. Decompose a conditional

- Conditional logic is a primary source of complexity
- The conditional code shows me what is happening but it doesn't tell me why
- It shortens a function and makes it easier to understand when we extract the conditional logic to a function that has a meaningful name
- This is actually a very specific case of Extract Function
- We can do this for each alternative in a series of if-elif-else statements
- https://sourcemaking.com/refactoring/decompose-conditional

3a. Replace Conditionals with Guard Clauses

- Conditional expressions come in two main styles:
  - In the first style, both legs of the conditional are part of normal behavior
  - In the second style, one leg is normal and the other indicates an unusual condition
- If both are part of normal behavior, we should use a condition with an if and an else
- If the condition is an unusual condition, we check for condition and return True or False
- This kind of check is often called a guard clause.
- https://refactoring.com/catalog/replaceNestedConditionalWithGuardClauses.html

3b. Replace Conditionals with Guard Clauses

```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```



```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

That's it for week 12!

We're all caught up!
Not so tough was it!
Work on A4!