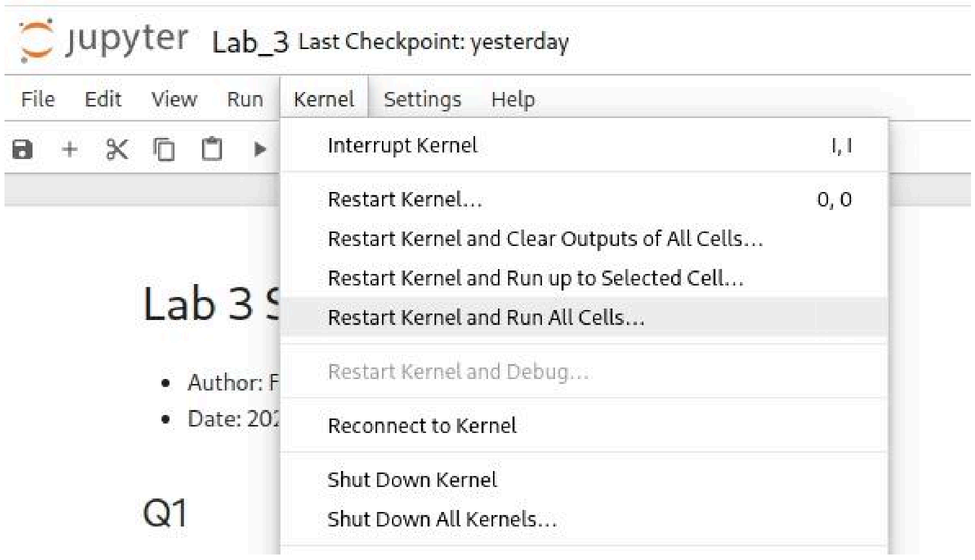
Lab 3: While, Modules, and Namespaces

Part 0: Submitting clean notebooks

As discussed yesterday, a clean well documented notebook is an important way to produce high quality reproducible scientific analyses. Like last week you will submit your code as a notebook but with some additional formatting requirements:

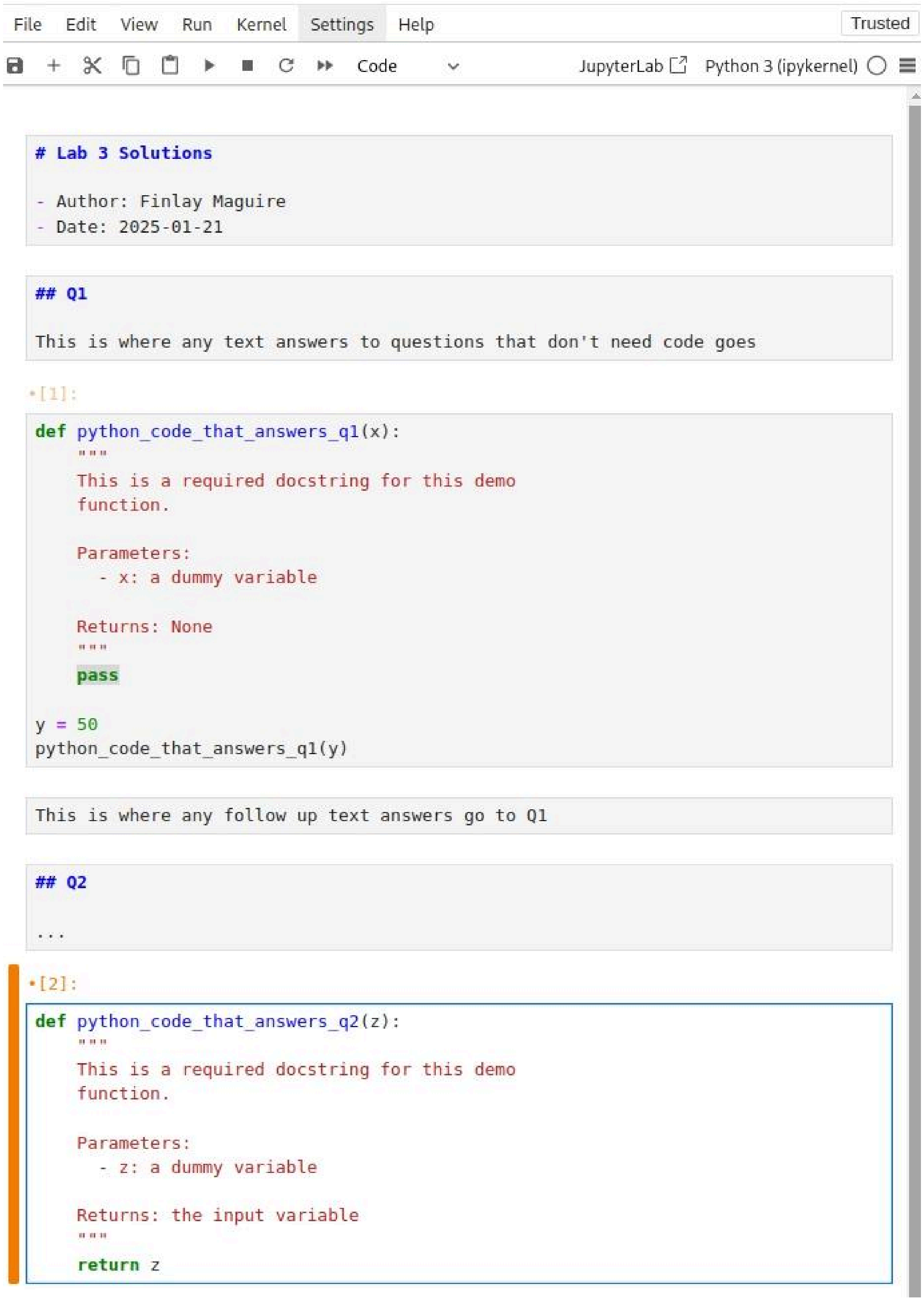
- The first cell should be a markdown cell containing appropriately formatted title (as heading 1), your name, and the date.
- Each answer should be preceded by a markdown cell that indicates the question (as heading 2).
- Every function should have a clear docstring that includes an explanation of what the function does, the parameters it takes in, and what it returns
- The notebook should be freshly run from top to bottom before submitting and any errors resolved (i.e., each code cell with executed in order - you can do this with Jupyter's "Kernel" Menu at the top -> "Restart Kernel and Run All Cells")



jupyter
Lab_3 Last Checkpoint: yesterday
File
Edit
View
Run
Kernel
Settings
Help
Interrupt Kernel I, I
Restart Kernel... 0, 0
Restart Kernel and Clear Outputs of All Cells...
Restart Kernel and Run up to Selected Cell...
Restart Kernel and Run All Cells...
Restart Kernel and Debug...
Reconnect to Kernel
Shut Down Kernel
Shut Down All Kernels...
Lab 3 S
Author: F
Date: 202
Q1

Markdown basic syntax can be found here: https://www.markdownguide.org/basic-syntax/ and remember you can change a cell from Code to Markdown (and back) using the down-arrow dropdown menu on the second row at the top.

Here is an example of a well-formatted notebook showing the markdown:



```
# Lab 3 Solutions

- Author: Finlay Maguire
- Date: 2025-01-21
```

```
## Q1

This is where any text answers to questions that don't need code goes
```

[1]:

```python
def python_code_that_answers_q1(x):
    """
    This is a required docstring for this demo
    function.

    Parameters:
      - x: a dummy variable

    Returns: None
    """
    pass

y = 50
python_code_that_answers_q1(y)
```

```
This is where any follow up text answers go to Q1
```

```
## Q2

...
```

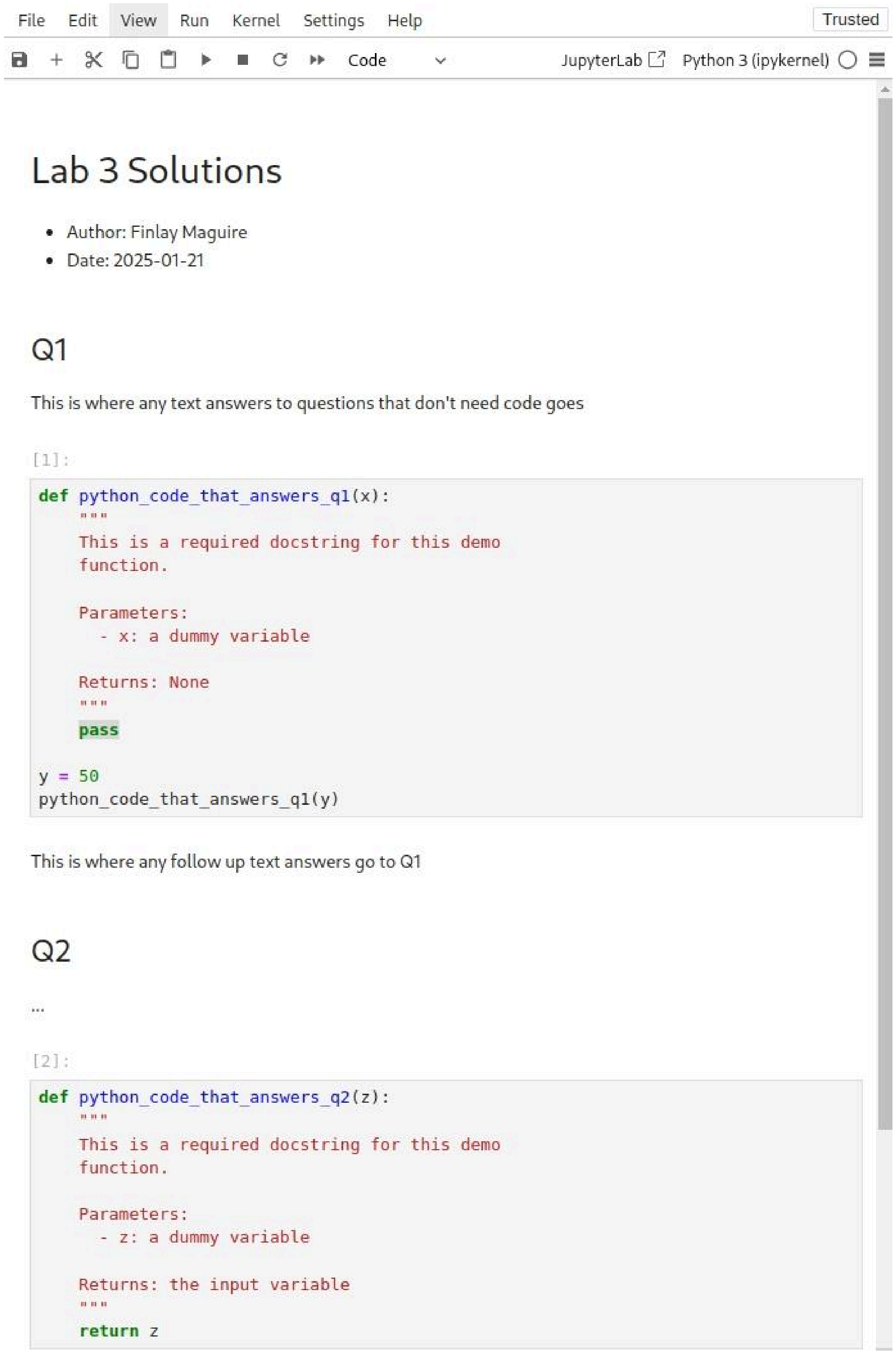
[2]:

```python
def python_code_that_answers_q2(z):
    """
    This is a required docstring for this demo
    function.

    Parameters:
      - z: a dummy variable

    Returns: the input variable
    """
    return z
```

And here is what it looks like when you run everything:



Lab 3 Solutions

- Author: Finlay Maguire
- Date: 2025-01-21

Q1

This is where any text answers to questions that don't need code goes

[1]:

```python
def python_code_that_answers_q1(x):
    """
    This is a required docstring for this demo
    function.

    Parameters:
      - x: a dummy variable

    Returns: None
    """
    pass

y = 50
python_code_that_answers_q1(y)
```

This is where any follow up text answers go to Q1

Q2

...

[2]:

```python
def python_code_that_answers_q2(z):
    """
    This is a required docstring for this demo
    function.

    Parameters:
      - z: a dummy variable

    Returns: the input variable
    """
    return z
```

Part 1: While Loops

Sometimes we want to loop until a certain conditional is no longer True. For example, let's say we have some boiling water at 100C and a room that is 25C. We hypothesise that this water will cool down each minute by 10% of its temperature difference with the room. How can we easily work out how long the water will cool down to room temperature? We can simulate it!

How do we do this? Well, we want to take the starting water temperature (100C), work out the difference between that and room temperature (25C), and then remove 10% of that value from the water temperature. Then we want to repeat this whole process UNTIL the water reaches room temperature.

The way we do this in python is by using a while-loop. What they do is simply do a test like an `if-statement`, but instead of running the code block *once*, they jump back to the "top" where the `while` is, and repeat. A `while-loop` runs until the expression is `False`.

Here's the problem with `while-loops`: Sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise you almost always want your loops to end eventually.

To avoid these problems, there are some rules to follow:

1. Make sure that you use `while-loops` sparingly. Usually a `for-loop` is better.
2. Review your while statements and make sure that the boolean test will become `False` at some point.
3. When in doubt, print out your test variable at the top and bottom of the `while-loop` to see what it's doing.

So, what would our water example look like:

```python
def cool_water(initial_temp, room_temp):
    """
    Simulates water cooling to room temperature.
    Each minute, the water loses 10% of its temperature difference
    with the room.

    Parameters:
    - initial_temp: Starting water temperature (Celsius)
    - room_temp: Room temperature (Celsius)

    Returns: Number of minutes to cool within 0.5 degrees of room temp
    """
    current_temp = initial_temp
    minutes = 0

    # Continue until within 0.5°C
    while abs(current_temp - room_temp) > 0.5:
        temp_difference = current_temp - room_temp
        # Lose 10%
        current_temp = current_temp - (0.1 * temp_difference)
        minutes += 1

    return minutes, current_temp
```

Q1. Radioactive decay is the process by which an unstable radioactive isotope turns into another isotope due to losing energy from emitting radiation. This is a random process but in aggregate we can express how quickly this happens in terms of “half-life” or a decay constant. Carbon-14 undergoes radioactive decay with a half-life of ~5,730 years. This means every 5,730 years half of your Carbon-14 will decay into other isotopes. Write a function with the parameters “initial_amount” (of radioisotope in grams) and half_life (in years) and then calculate how many half-lives (or equivalent years) it will be until you have <= 1g of a radioisotope left.

Part 2: Dictionaries

Dictionaries are a way of representing `key=value` data, we use this type of data all the time without realizing it. When you read an email you might have:

```
From: j.smith@example.com
To: zed.shaw@example.com
Subject: I HAVE AN AMAZING INVESTMENT FOR YOU!!!
```

On the left are the keys (From, To, Subject) which are *mapped* to the contents on the right of the `:`. Programmers say the key is "mapped" to the value, but they could also say "set to". As in, "I set `From` to `j.smith@example.com`." In Python I might write this same email using a dictionary like this:

```
email = {
    "From": "j.smith@example.com",
    "To": "zed.shaw@example.com",
    "Subject": "I HAVE AN AMAZING INVESTMENT FOR YOU!!!"
};
```

You create a dictionary by:

1. Opening it with a `{` (curly-brace).
2. Writing the key, which is a string here, but can be numbers, or almost anything.
3. Writing a `:` (colon).
4. Writing the value, which can be anything that's valid in Python.

Once you do that, you can access this Python email like this:

```
email["From"]
'j.smith@example.com'

email["To"]
'zed.shaw@example.com'

email["Subject"]
'I HAVE AN AMAZING INVESTMENT FOR YOU!!!'
```

The only difference from `list` indexes is that you use a string (`'From'`) instead of an integer. Just like a list, the index and items can also be variables. Dictionaries can use a string, boolean, integer, or float as a key (or variables containing them) but not a list or a tuple. You can also put a dictionary inside a list or a list inside a dictionary (as a value).

```
list_of_dictionaries = [
     {'a': 5},
     {1: 'a', 2: 'b'},
     {True: x, False: y},
]

dictionary_of_lists = {'a': [1, 2, 3], 1: ['a', 'b', 'c']}
```

You are now going to repeat the exercise we did last week with `lists` to get some practice accessing dictionaries.

Q2. Copy the following dictionaries into your notebooks. For each of the following `lists` of `dictionaries` and pieces of data write down how to get that information. You should attempt to do this in your head by looking at the code, then test your guess in the Jupyter

```
fruit = [
    {'kind': 'Apples',  'count': 12, 'rating': 'AAA'},
    {'kind': 'Oranges', 'count': 1,  'rating': 'B'},
    {'kind': 'Pears',   'count': 2,  'rating': 'A'},
    {'kind': 'Grapes',  'count': 14, 'rating': 'UR'}
];

languages = [
    {'name': 'Python', 'speed': 'Slow',
     'opinion': ['Terrible', 'Mush']},
    {'name': 'JavaScript', 'speed': 'Moderate',
     'opinion': ['Alright', 'Bizarre']},
    {'name': 'Perl6', 'speed': 'Moderate',
     'opinion': ['Fun', 'Weird']},
    {'name': 'C', 'speed': 'Fast',
     'opinion': ['Annoying', 'Dangerous']},
    {'name': 'Forth', 'speed': 'Fast',
     'opinion': ['Fun', 'Difficult']},
];
```

You need to get all of these elements out of the `fruit` variable:

- 12
- 'AAA'
- 2
- 'Oranges'
- 'Grapes'
- 14
- 'Apples'

You need to get all of these elements out of the `languages` variable:

- 'Slow'
- 'Alright'
- 'Dangerous'
- 'Fast'
- 'Difficult'
- 'Fun'
- 'Annoying'
- 'Weird'
- 'Moderate'

Q3. Write a function that takes the following dictionary and returns a new dictionary where the key and values have been swapped i.e.,3389.5 is now the key for the value "radius_km"

```
mars_data = {
    'mass_kg': 6.39e23,
    'radius_km': 3389.5,
    'density_g_cm3': 3.93,
    'surface_gravity_m_s2': 3.71,
    'escape_velocity_km_s': 5.03,
    'orbit_semi_major_axis_AU': 1.524,
    'orbit_period_days': 687,
    'orbit_eccentricity': 0.0934,
    'orbit_inclination_deg': 1.85,
    'atmosphere_co2_percent': 95.3,
    'atmosphere_n2_percent': 2.7,
    'atmosphere_ar_percent': 1.6,
    'atmosphere_o2_percent': 0.13,
    'atmosphere_pressure_kPa': 0.636,
    'average_temp_K': 210,
}
```

Q4. Write a function that uses a dictionary which translates a DNA string to an RNA string (and returns it) by replacing ‘A’ with ‘U’, ‘G’ with ‘C’ ,’C’ with ‘G’, and ‘T’ with ‘A’ i.e., ‘AGCT’ would become ‘UCGA’

Part 3. Modules and Packages[1]

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module’s name (as a string) is available as the value of the global variable `__name__`. For example, you could create a file called `fibo.py` in the current directory with the following contents:

```
# Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Now enter your notebook and import this module with the following command:

```
import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current namespace; it only adds the module name `fibo` there. Using the module name you can access the module namespace to use the functions:

```
fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

If you intend to use a function often you can assign it to a local name:

[1] Following explanation is modified from the excellent Python documentation https://docs.python.org/3/tutorial/modules.html

```
fib = fibo.fib

fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
from fibo import fib, fib2
fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
from fibo import *
fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
import fibo as fib
fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fib`.

It can also be used when utilising from with similar effects:

```
from fibo import fib as fibonacci
fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
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
              ...
```

When importing the package, Python searches through the directories in PYTHONPATH (i.e., `sys.path`) looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a namespace package, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
import sound.effects.echo
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Q5. Complete this function and put it in a file called lab_functions.py in the same folder as your notebook

```python
# lab_functions.py
def smallest_number(list_of_numbers):
      """The functions takes a list of numbers (integers or floats) and
      finds the smallest number and returns using only a for loop
      and if statement"""
      pass
```

Then in your notebook import that function using `import lab_functions` and show the output for `lab_functions.smallest_number([54, 100, 12]).`

Q6. In maths we often represent things using an infinite sum (also known as a series). For example, we can calculate the value of π using the following expression.

$$\pi = 4 * \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$$

In python we can't actually do something infinitely! Write a function with a docstring that calculates and returns the value of π using the series above to a user specified number of terms (i.e., if we stopped at the + … above then it would to n=4). Then import and execute this function for n=100.

Q7. Create a package called "lab_package" that contains a small.py module with the function from Q5 and a pi.py containing the function from Q6. Demonstrate you have done this correctly by importing both functions and executing them e.g., lab_package.small.smallest_number(list(range(100))

Part 4: Namespaces[2]

Roughly speaking, namespaces are just containers for mapping names to data objects (like a specific variable, list, dictionary, lazy enumerate/zip/range, or function definition). Such a "name-to-object" mapping allows us to access an object by a name that we've assigned to it. E.g., if we make a simple string assignment via `a_string = "Hello string"`, we create a reference to the `"Hello string"` object, and henceforth we can access it via its variable name `a_string`.

We can picture a namespace as a Python dictionary structure, where the dictionary keys represent the names and the dictionary values the object itself (and this is also how namespaces are currently implemented in Python), e.g.,

```python
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
```

Now, the tricky part is that we have multiple independent namespaces in Python, and names can be reused in different namespaces (only the objects are unique), for example:

```python
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
b_namespace = {'name_a':object_3, 'name_b':object_4, ...}
```

[2] Drawn on https://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html

So namespaces can exist independently from each other and that they are structured in a certain hierarchy, which brings us to the concept of “scope”. The “scope” in Python defines the “hierarchy level” in which we search namespaces for certain “name-to-object” mappings.

For example, let us consider the following code:

```
i = 1
def foo():
    i = 5
    print(i, 'in foo()')
print(i, 'global')
foo()

# output
1 global
5 in foo()
```

Here, we just defined the variable name `i` twice, once on the `foo` function.

- `foo_namespace = {'i':object_3, ...}`
- `global_namespace = {'i':object_1, 'name_b':object_2, ...}`

So, how does Python know which namespace it has to search if we want to print the value of the variable `i`? This is where scope comes into play.

Multiple namespaces can exist independently from each other and that they can contain the same variable names on different hierarchy levels. The “**scope**” defines on which hierarchy level Python searches for a particular “name” to get its associated object. Now, the next question is: “In which order does Python search the different levels of namespaces before it finds the ‘name-to-object’ mapping?

To answer is: It uses the LEGB-rule, which stands for

Local -> Enclosed -> Global -> Built-in,

where the arrows should denote the direction of the namespace-hierarchy search order.

- *Local* is the “bottom” level; this would be the inside of a simple function or the innermost (deepest/most-indented) function if a function is defined in a function.
- *Enclosed* if we have a function defined inside a function then the enclosed namespace would be anything defined in the outer functions. This namespace doesn’t always exist, it only exists if we need extra namespace layers between the local and the global (such as with a nested function: top level/no-indent is *global*, then function definition is *enclosed*, and a function on the inside of that is *local*).
- *Global* refers to the uppermost level of the executing script itself i.e., the code that isn’t indented.
- *Built-in* are special names that Python reserves for itself.

So, if a particular name:object mapping cannot be found in the local namespaces, the namespaces of the enclosed scope will be searched next. If the search in the enclosed scope is unsuccessful, too, Python moves on to the global namespace, and eventually, it will search the built-in namespace (sidenote: if a name cannot be found in any of the namespaces, a *NameError* will be raised).

Namespaces can also be further nested, for example if we import modules. In those cases we have to use prefixes to access those nested namespaces. Let me illustrate this concept in the following code block:

```
import numpy
import math
import scipy

print(math.pi, 'from the math module')
print(numpy.pi, 'from the numpy package')
print(scipy.pi, 'from the scipy package')

3.141592653589793 from the math module
3.141592653589793 from the numpy package
3.141592653589793 from the scipy package
```

This is also why we have to be careful if we import modules via "`from a_module import *`", since it loads the variable names into the global namespace and could potentially overwrite already existing variable names.

Q8. Explain what each of the following bits of code will output and why in terms of built-in, global, nested, and local namespaces:

```
x = 10
def print_x():
    x = 5
    print(f"Local x: {x}")

print_x()
print(f"Global x: {x}")
```

```
x = ['1a', 'b', 'c']
print(len(x))
def len(x):
    return 42

print(len(x))
```

```python
a_var = 'global value'

def outer():
    a_var = 'enclosed value'
    def inner():
        a_var = 'local value'
        print(a_var)
    inner()

outer()
```