Example 1: Signal Processing with Python in Jupyter Notebooks

This example is a Jupyter Notebook (teaching notebook/problem set) that that illustrates how I use a fun but effective narrative to interactively teach simple programming concepts. This is typical of how I used interactive activities in my classroom. In particular, my strategy here is to separate the programming concepts from the science concept, to avoid unnecessary confusion. This can be seen in the following pages, which are created from three files:

1. The README file provided to students explaining the assignment
2. The Programming Assignment, a Jupyter Notebook that introduces concepts like iterables and stem plots
3. The Workshop Assignment, a Jupyter Notebook that uses iterables and stem plots to interpret and evaluate a spectral analysis using the Discrete Fourier Transform.

Note that both notebook files are actually "solution" notebooks: the solutions are included in the code, and the code provided to the students is commented in the code cells (typically with missing parts to fill in).

You can read the assignments in a browser or download the notebook files using the following links:

1. README: **read in a browser**
2. Programming Assignment: read in a browser as **student** or **solution** versions, or download the notebooks as **student** or **solution** versions.
3. Workshop Assignment: read in a browser as **student** or **solution** versions, or download the notebooks as **student** or **solution** versions.

Note that the notebooks are compatible with most versions of Python 3.X and a virtual environment containing numpy and matplotlib.

The documents provided here are explicitly shared here as part of the portfolio of Robert Lanzafame. We are currently working on finalizing the open source licenses, so please contact Robert at R.C.Lanzafame@tudelft.nl if you are interested in using them.

README for PA 2.3: iTer-remoto

*CEGM1000 MUDE: Week 2.3. Due: complete this PA prior to class on **Wednesday**, Nov 27!).*

You can access this assignment with the following link: classroom.github.com/a/MbJc5oCJ.

This assignment will address a few simple, but useful, topics related to indexing and iterating in Python, illustrated with an annotated stem figure. The content of this PA is simple, but will be used directly in the **Wednesday** workshop this week.

Specifically, we will look at:

- iterables and iterable objects
- iterables `range`, `enumerate` and `zip`
- the modulo operator `%`
- plot type `stem`

Note that the Wednesday workshop will use stem plots, so we strongly encourage you to do this early in the week---but only if you have time after covering the DFT chapter in the book!

There are three files in this PA:

- this `README.md`
- a notebook `PA_2_3_iter_remoto.ipynb`
- an image `earthquakes.svg`, which you are asked to duplicate

Grading Criteria

You will pass this PA if:

1. Your notebook `PA_2_3_iter_remoto.ipynb` runs without errors.
2. After running your notebook, a file `my_earthquake.svg` is created in the repository that contains your version of the image.

End of file.

PA 2.3: iTer-remoto

CEGM1000 MUDE: Week 2.3. Due: complete this PA prior to class on Wednesday, Nov 27!).

Note that the Wednesday workshop will use stem plots, so we strongly encourage you to do this early in the week!

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Introduction

Suppose we are writing code to summarize some simple data, perhaps a group of teams that each have a score for a particular game.

In [2]:
```
team = ['green', 'red', 'blue']
score = [5, 9, 7]
```

Suppose we want to list the names of each team. One typical approach is to loop over each item in the list:

In [3]:
```
N_teams = len(team)
print('The team names are:')
for i in range(N_teams):
    print(f'{team[i]}')
```

```
The team names are:
green
red
blue
```

However, this is not the most efficient way to do this, as we need to define two extra variables: `N_teams` and the iteration index `i`. There is a better way!

Iterable Objects

Python supports what is referred to as *iterable* objects: generally, this means objects that can be easily used by Python's built-in and efficient iteration schemes. Examples are: list, tuple, dictionaries, and strings. You may be interested in reading about all of the technical details here; however, for this assignment the explanations here and examples below are sufficient. It is important to recognize this type of object exists in Python, so we can take advantage of it in our code, or so you can recognize the algorithms in other authors code.

Let's see an iterable object in action. First, an easy way to test if an object is iterable is to use it as an argument of the `iter()` method:

In [4]:
```
iter(team)
```

Out[4]:
```
<list_iterator at 0x1e90e230400>
```

Other objects can also be made into an iterator, like numpy arrays:

In [5]:
```
iter(np.array([1, 5, 67]))
```

Out[5]:
```
<iterator at 0x1e90f3e5780>
```

The cell above should return `iterator`, which indicates the object (argument) is iterable. An integer, however, is *not* iterable.

Task 1:

Try running the code below to confirm that an integer is not iterable (an error will occur), then fix it by converting the argument to an iterable object. Experiment by turning the integer into a list, string and np.array.

In [6]:
```
# iter(5)

# SOLUTION
iter([5])
iter('5')
iter(np.array([5]))
```

Out[6]:
```
<iterator at 0x1e90e07b970>
```

Great, you can make an iterator! But what do you *do* with it?

Most of the time we simply use it in a `for` *loop, but it is worthwhile to understand this object a bit more deeply.*

One simple way to understand an iterator is to imagine that it is a way of (efficiently!) turning your iterable object into something that can go sequentially through all its values. To do this it has two "abilities":

1. the iterator knows its current value, and
2. the iterator knows its next value

Let's try using the iterator by running the following cells:

In [7]:
```
letters = 'abcde'
letter_iterator = iter(letters)
```

Nothing new yet, but now watch what happens when we use the Python built-in command `next()`

Run the cell multiple times.

In [8]:
```
next(letter_iterator)
```

Out[8]:
```
'a'
```

There are two key things to note here:

1. The iterator "knows" how to go through all of its values, and
2. Calls to `next` returns to value itself, *not the index!*

This is a very useful feature, as we can see in this simple example below:

In [9]:
```
for i in letters:
    print(i)
```

```
a
b
c
d
e
```

Obviously this is a simple `for` loop, but hopefully the explanation above indicates what is actually happening "under the hood" of the `for` loop:

1. identifies `letters` as an iterable object
2. converts it into an iterator, and then it
3. finds the next value in the sequence and assigns it to variable `i`
4. stop once there is no next value

This is a bit of a simplification of what happens in reality, but it gets the idea across.

Now let's take a look at something you have probably used a lot: the `range()` method:

In [10]:
```
print(type(range(5)))
print(range(5))
```

```
<class 'range'>
range(0, 5)
```

Although it looks like it produces a tuple, `range` is a special iterable object that simply counts. As we can see it works the same as our `for` loop above:

In [11]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Hopefully this explains why `range` is such a useful feature for accessing sequential indexed elements of arrays.

It turns out there are two more built-in Python methods that produce useful iterable objects: `enumerate` and `zip`. Let's take a look ath their doc strings.

Task 2:

Read the docstrings below for the two methods. Confirm that they produce an iterable, but compare them to observe how the input and output of each is different. Can you imagine why one could be more useful than another in certain situations?

`enumerate`:

```
Init signature: enumerate(iterable, start=0)
Docstring:
Return an enumerate object.

  iterable
    an object supporting iteration

The enumerate object yields pairs containing a count (from start, which
defaults to zero) and a value yielded by the iterable argument.

enumerate is useful for obtaining an indexed list:
    (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
```

And `zip`:

```
Init signature: zip(self, /, *args, **kwargs)
Docstring:
zip(*iterables, strict=False) --> Yield tuples until an input is exhausted.

   >>> list(zip('abcdefg', range(3), range(4)))
   [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]

The zip object yields n-length tuples, where n is the number of iterables
passed as positional arguments to zip().  The i-th element in every tuple
comes from the i-th iterable argument to zip().  This continues until the
shortest argument is exhausted.

If strict is true and one of the arguments is exhausted before the others,
raise a ValueError.
```

The main takeaways, should be as follows:

1. Yes, they both produce an iterator
2. `enumerate` takes one iterable object and returns indices along with the value
3. `zip` takes two iterable objects and returns their values

Let's try them out by running the next cell. Does it behave as expected?

In [12]:
```
thing_1 = 'roberts_string'
thing_2 = [2, 3, 36, 3., 1., 's', 7, '3']

test_1 = enumerate(thing_1)
print(f'We created: {test_1}')
print(next(test_1), next(test_1), next(test_1))

test_2 = zip(thing_1, thing_2)
print(f'We created: {test_2}')
print(next(test_2), next(test_2), next(test_2))
```

```
We created: <enumerate object at 0x000001E90F50A070>
(0, 'r') (1, 'o') (2, 'b')
We created: <zip object at 0x000001E90F519F40>
('r', 2) ('o', 3) ('b', 36)
```

Can you see the difference?

Looking at them in a for loop will also illustrate what's going on:

In [13]:
```
print('First, enumerate:')
for i, j in enumerate(thing_1):
    print(i, j)
print('\nThen, zip:')
for i, j in zip(thing_1, thing_2):
    print(i, j)
```

```
First, enumerate:
0 r
1 o
2 b
3 e
4 r
5 t
6 s
7 _
8 s
9 t
10 r
11 i
12 n
13 g

Then, zip:
r 2
o 3
b 36
e 3.0
r 1.0
t s
s 7
_ 3
```

Now let's return to our teams from the beginning of this assignment: let's apply our knowledge of `enumerate` and `zip` to see if we can print out the points per team in an efficient way.

Task 3:

Use `enumerate` to print out the summary of points per team according to the print statement.

Hint: you only need to use one of the lists as an argument.

In [14]:
```
# team = ['green', 'red', 'blue']
# score = [5, 9, 7]

# for YOUR_CODE_WITH_enumerate_HERE:
#     print(f'Team {} has {} points.')

# SOLUTION
team = ['green', 'red', 'blue']
score = [5, 9, 7]

for i, j in enumerate(score):
    print(f'Team {team[i]} has {j} points.')
```

```
Team green has 5 points.
Team red has 9 points.
Team blue has 7 points.
```

You may have noticed that `enumerate` is a bit awkward for this case, since we still need to define an unnecessary iteration index to access the team name. Let's see if `zip` makes things easier:

Task 4:

Use `zip` to print out the summary of points per team according to the print statement.

In [15]:
```
# team = ['green', 'red', 'blue']
# score = [5, 9, 7]
```

```
# for YOUR_CODE_WITH_zip_HERE:
#     print(f'Team {} has {} points.')

# SOLUTION
team = ['green', 'red', 'blue']
score = [5, 9, 7]

for i, j in zip(team, score):
    print(f'Team {i} has {j} points.')
```

```
Team green has 5 points.
Team red has 9 points.
Team blue has 7 points.
```

Task 7:

Using the tools described in this assignment, recreate the plot above. Note that to avoid cluttering the figure, the detailed earthquake information is only plotted for years that are a multiple of 5 (i.e., 2000, 2005, etc.).

That's really compact!

Modulo Operator `%`

The modulo operator `%` is common in programming languanges, but not so much in our typical engineering problems. It turns out it can be very useful in iteration loops. It's actually quite simple: *the expression* `a % b` *returns the remainder of* `a/b`. That's it!

Take a look at the folloiwing examples that illustrate the operator:

In [16]:
```
print(6 % 5)
print(5 % 6)
print(1 % 10)
print(5 % 5)
print(0 % 5)
```

```
1
5
1
0
0
```

You can't divide by zero, just as in normal division.

Below is an illustration for how the modulo is useful for doing things in a `for` loop that doesn't need to be done on every iteration.

In [17]:
```
for i in range(100):
    if i%25 == 0:
        print(i)
```

```
0
25
50
75
```

Task 5:

Experiment with the two Python cells above and make sure you understand what the modulo operator is and how it works.

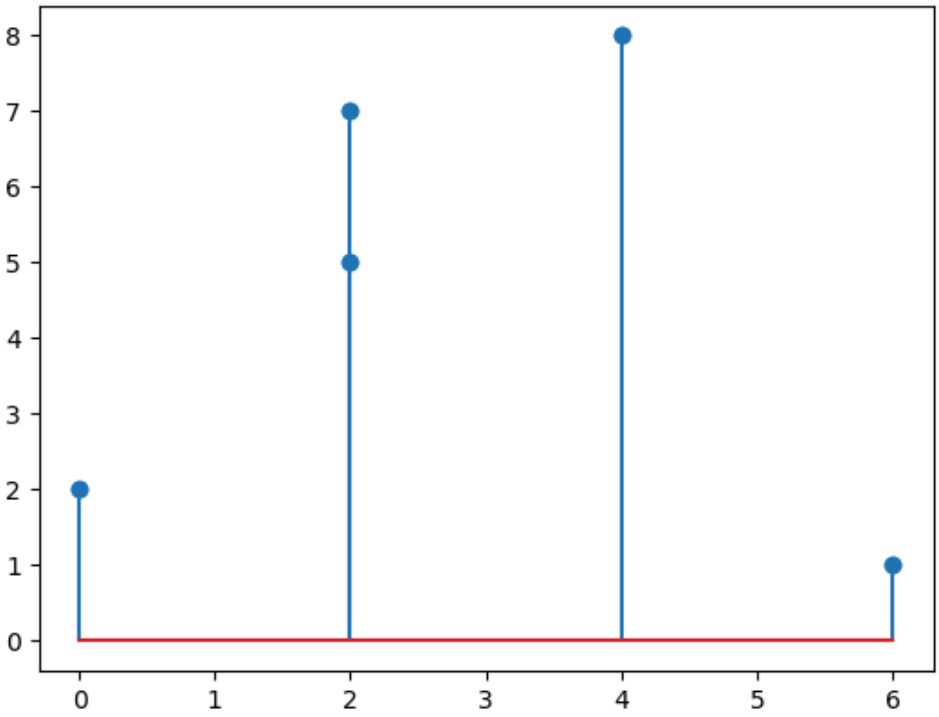
We won't do anything else with `%` now, but will apply this in the last task of the assignment.

Stem Plot Type

You have probably used the matplotlib plot types `plot`, `hist` and `scatter` frequently; another type is `stem`, which is useful for indicating the magnitude of various points along a number line. As with a scatter plot, the data does not need to be ordered, and missing values are easy to handle.

> **Task 6:**
>
> Run the cell below and play with the values to make sure you are comfortable with the stem plot type. Do you see how it can handle two values with the same index?

In [18]:
```
value = [2, 7, 5, 1, 8]
index = [0, 2, 2, 6, 4]
plt.plot(index, value, 'o')
plt.stem(index, value);
```



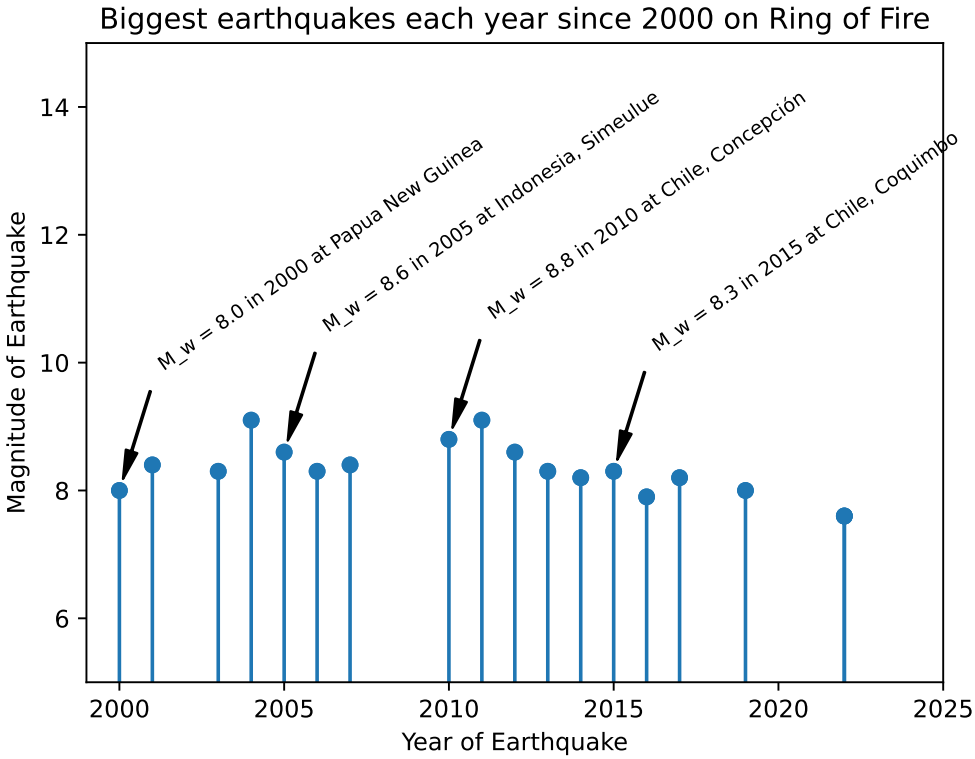
Bringing it all together: Earthquake analysis

To put the ideas above into practice, consider the following data, in the form of three lists, that describes the largest earthquakes in the world since 2025, that are also located on the ring of fire (Source: Wikipedia).

In [20]:
```
year = [2003, 2011, 2013,
        2006, 2022,
        2017, 2019, 2001, 2010, 2015,
        2014, 2022, 2016,
        2000, 2007, 2012, 2005, 2004]

magn = [8.3, 9.1, 8.3,
        8.3, 7.6,
        8.2, 8.0, 8.4, 8.8, 8.3,
        8.2, 7.6, 7.9,
        8.0, 8.4, 8.6, 8.6, 9.1]

site = ['Japan, Hokkaidō',
        'Japan, Honshu',
```

```
        'Russia, Sea of Okhotsk',
        'Russia, Kuril Islands',
        'Mexico, Michoacán',
        'Mexico, Chiapas',
        'Peru, Loreto',
        'Peru, Arequipa',
        'Chile, Concepción',
        'Chile, Coquimbo',
        'Chile, Iquique',
        'Papua New Guinea, Morobe',
        'Papua New Guinea, New Ireland',
        'Papua New Guinea',
        'Indonesia, Sumatra',
        'Indonesia, Indian Ocean',
        'Indonesia, Simeulue',
        'Indonesia, Sumatra']
```

We can visualize the data in the following figure. Can you find an easy way to recreate this plot?



Task 7:

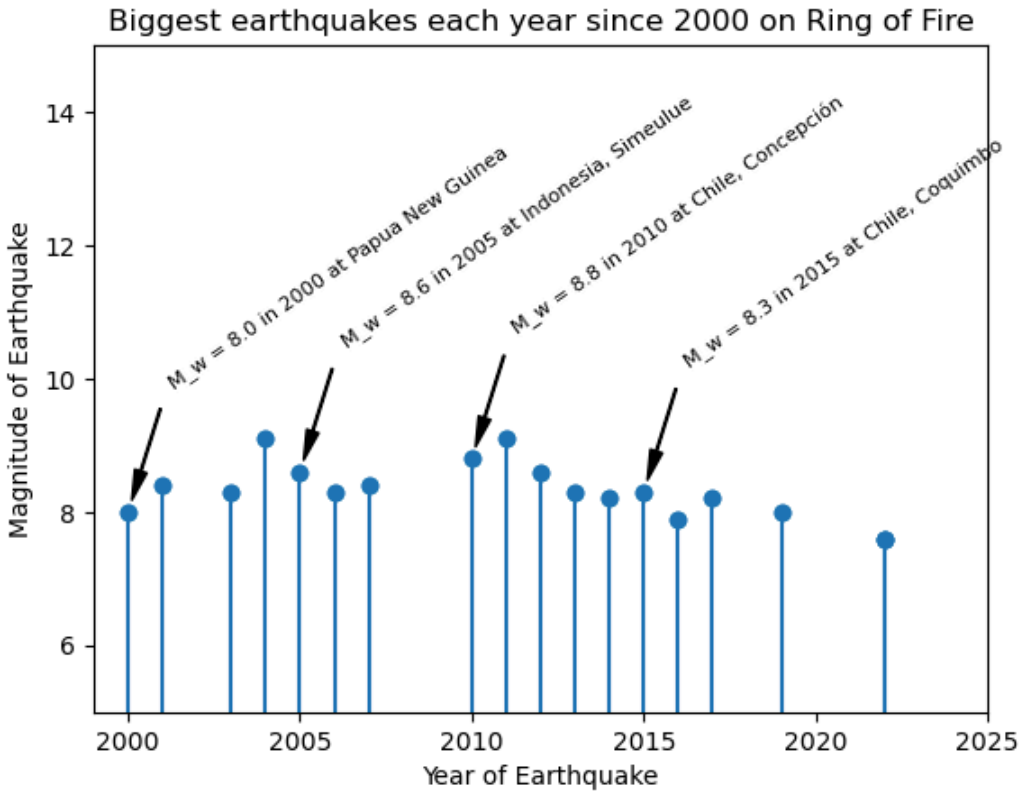
Using the tools described in this assignment, recreate the plot above. Note that to avoid cluttering the figure, the detailed earthquake information is only plotted for years that are a multiple of 5 (i.e., 2000, 2005, etc.).

The code cell below already contains the required code to generate the svg file.

Hints: create a for loop using 3 iterables (`x` , `y` and `z`) based on *one* of these iterable tools: `range` , `enumerate` or `zip` . Use the modulo operator `%` to choose the 5-year multiples. An outline of the solution in pseudo-code is also provided here, each of which should be one line in your code (except item 5):

```
1  for each iterable item
2     if the year is a multiple of 5
3         create the label (string)
4         annotate the plot (i.e., annotate(...))
5  include any additional formatting (more than one line)
```

In [21]:
```
plt.plot(year, magn, 'o')
plt.stem(year, magn)


for x, y, z in zip(year, magn, site):
    if x%5 == 0:
        label = f"M_w = {y} in {x} at {z}"
        plt.annotate(label,
                     (x,y), # these are the coordinates to position the label
                     textcoords="offset points",
                     xytext=(15,50),
                     fontsize=8,
                     ha='left',
                     rotation=35,
                     arrowprops=dict(facecolor='black', shrink=0.1, width=0.5, headwidth=5))
plt.xlim(1999, 2025)
plt.ylim(5,15)
plt.xlabel('Year of Earthquake')
plt.ylabel('Magnitude of Earthquake')
plt.title('Biggest earthquakes each year since 2000 on Ring of Fire');
plt.savefig('my_earthquake.svg') # Don't remove this line
```



End of notebook.

WS 2.3: Discrete Fourier Transform (DFT): You Try Meow (Miauw)

CEGM1000 MUDE: Week 2.3, Signal Processing. For: November 27, 2024

The goal of this workshop to work with the *Discrete Fourier Transform* (DFT), implemented in Python as the *Fast Fourier Transform* (FFT) through `np.fft.fft`, and to understand and interpret its output.

The notebook consists of two parts:

- The first part (Task 0) is a demonstration of the use of the DFT (*you read and execute the code cells*),
- The second part is a simple exercise with the DFT (*you write the code*).

To start off, let's do a quick quiz question: *what is the primary purpose of the DFT?*

Find the answer here.

That's right! We convert our signal into the frequency domain. And if you would like an additional explanation of the key frequencies, you can find it here.

> Note the use of `zip`, `stem`, `annotate` and the modulo operator `%`. Refer to PA11 if you do not understand these tools. Furthermore, note that the term _modulus_ is also used here (and in the textbook), which is another term for _absolute value._

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
```

Task 0: Demonstration of DFT using pulse function

In the first part of this notebook, we use $x(t) = \Pi(\frac{t}{4})$, and its Fourier transform $X(f) = 4\,\mathrm{sinc}(4f)$, as an example (see the first worked example in Chapter 3 on the Fourier transform). The pulse lasts for 4 seconds in the time domain; for convenience, below it is not centered at t = 0, but shifted (delayed) to the right.

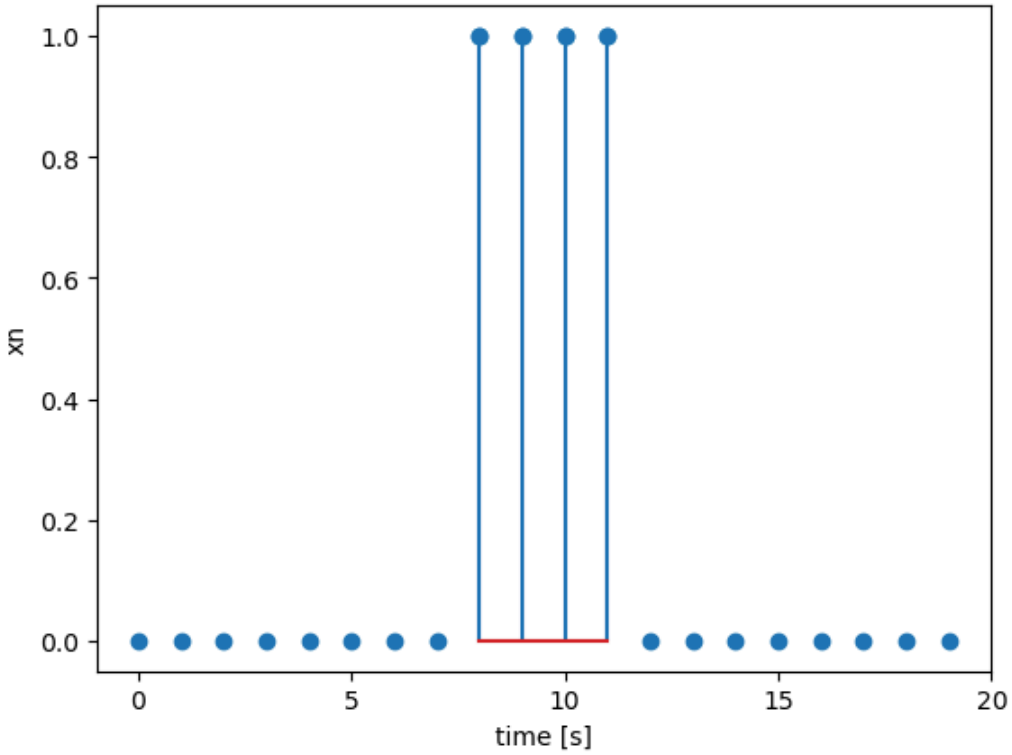
The signal $x(t)$ clearly is non-periodic and is an energy signal; apart from a short time span of 'activity' it is zero elsewhere.

We create a pulse function $x(t)$ in discrete time x_n by numpy. The total signal duration is T = 20 seconds (observation or record length). The sampling interval is Δt = 1 second. There are N = 20 samples, and each sample represents 1 second, hence $N\Delta t = T$.

Note that the time array starts at t = 0 s, and hence, the last sample is at t = 19 s (and not at t = 20 s, as then we would have 21 samples).

Task 0.1: Visualize the Signal

In [2]:
```
t = np.arange(0,20,1)
xt = np.concatenate((np.zeros(8), np.ones(4), np.zeros(8)))

plt.plot(t, xt,'o')
plt.stem(t[8:12], xt[8:12])
plt.xticks(ticks=np.arange(0,21,5), labels=np.arange(0,21,5))
plt.xlabel('time [s]')
plt.ylabel('xn');
```



Task 0.2: Evaluate (and visualize) the DFT

We use `numpy.fft.fft` to compute the one-dimensional Discrete Fourier Transform (DFT) of x_n, which takes a signal as argument and returns an array of coefficients (refer to the documentation as needed).

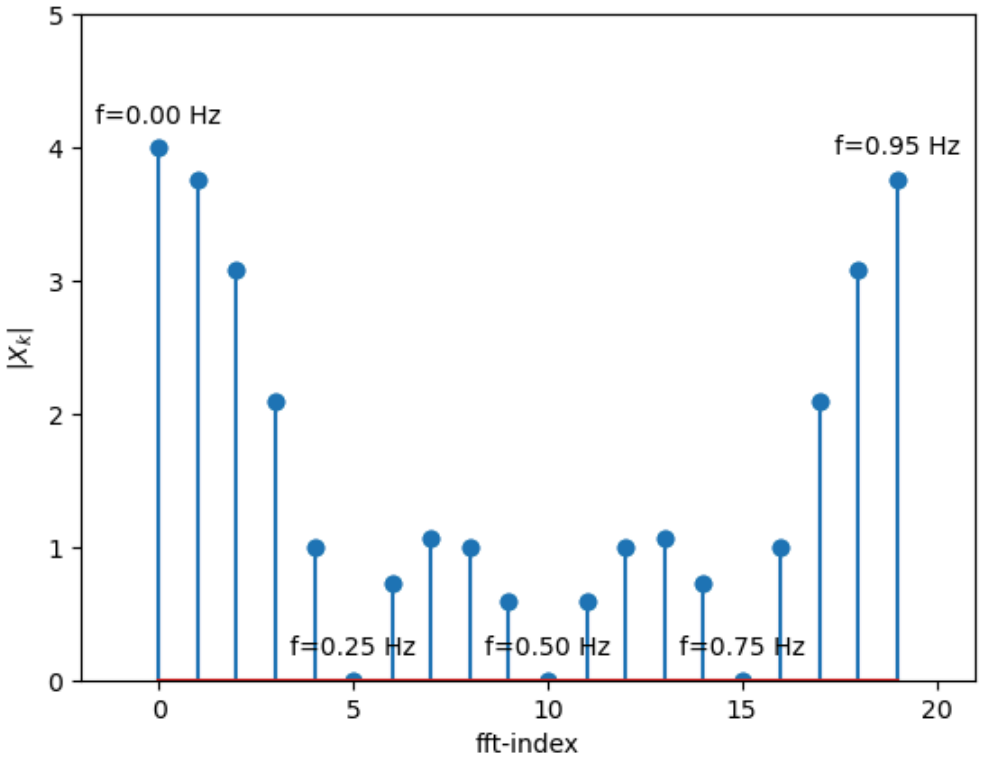
The DFT converts N samples of the time domain signal x_n, into N samples of the frequency domain. In this case, it produces X_k with $k = 0, 1, 2, \ldots, N-1$, with N = 20, which are complex numbers.

Task 0.2: Read the code cell below before executing it and identify the following:

1. Where is the DFT computed, and what is the output?
2. Why is the modulus (absolute value) used on the DFT output?
3. What are the values are used for the x and y axes of the plot?
4. How is frequency information found and added to the plot (mathematically)?

Once you understand the figure, continue reading to understand *what do these 20 complex numbers mean, and how should we interpret them?*

In [3]:
```
abs_fft = np.abs(np.fft.fft(xt))
index_fft = np.arange(0,20,1)
plt.plot(index_fft, abs_fft, 'o')

freq = np.arange(0, 1, 0.05)
for x,y in zip(index_fft, abs_fft):
    if x%5 == 0 or x==19:
        label = f"f={freq[x]:.2f} Hz"
        plt.annotate(label,
                     (x,y),
                     textcoords="offset points",
                     xytext=(0,10),
                     fontsize=10,
                     ha='center')
plt.ylim(0,5)
plt.xlim(-2,21)
plt.xlabel('fft-index')
plt.ylabel('$|X_k|$')
plt.stem(index_fft, abs_fft);
```

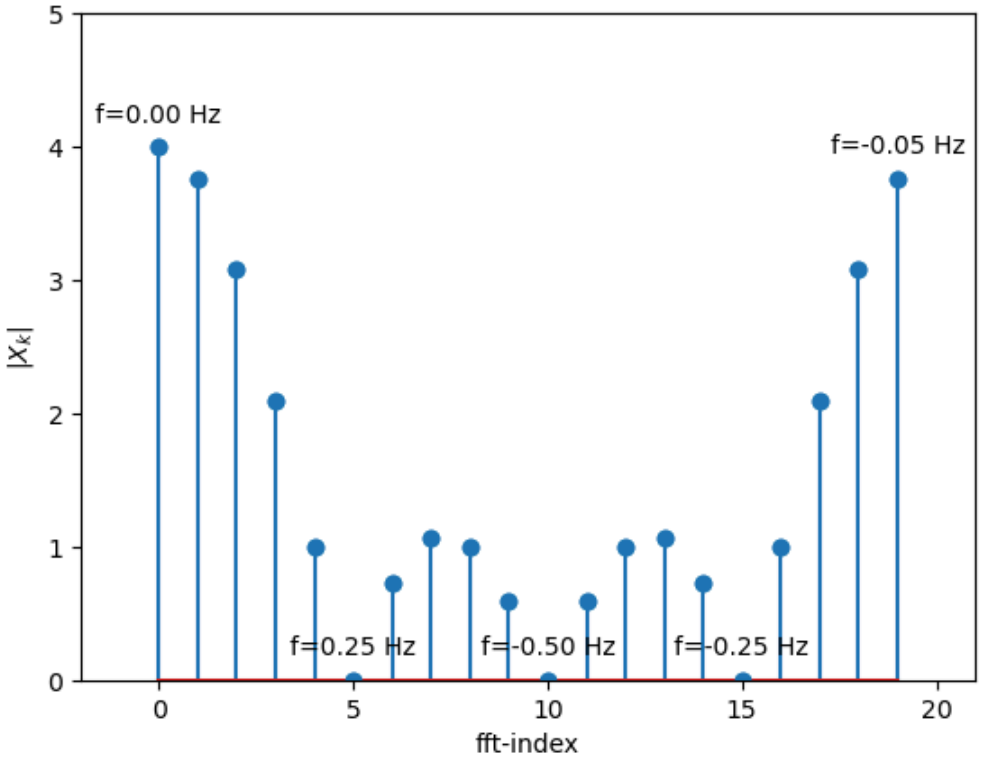


The frequency resolution Δf equals one-over-the-measurement-duration, hence $\Delta f = 1/T$. With that knowledge, we can reconstruct the frequencies expressed in Hertz.

The spectrum of the sampled signal is periodic in the sampling frequency f_s which equals 1 Hz (Δt = 1 s). Therefore, it is computed just for one period $[0, f_s)$. The last value, with index 19, represents the component with frequency f = 0.95 Hz. A spectrum is commonly presented and interpreted as double-sided, so in the above graph we can interpret the spectral components with indices 10 to 19 corresponding to negative frequencies.

Task 0.3: Identify negative frequencies

In [4]:
```
abs_fft = np.abs(np.fft.fft(xt))
plt.stem(index_fft, abs_fft)
plt.plot(index_fft, abs_fft, 'o')

freq = np.concatenate((np.arange(0, 0.5, 0.05), np.arange(-0.5, 0, 0.05)))
for x,y in zip(index_fft, abs_fft):
    if x%5 == 0 or x==19:
        label = f"f={freq[x]:.2f} Hz"
        plt.annotate(label,
                     (x,y),
                     textcoords="offset points",
                     xytext=(0,10),
                     fontsize=10,
                     ha='center')
plt.ylim(0,5)
plt.xlim(-2,21)
plt.xlabel('fft-index')
plt.ylabel('$|X_k|$');
```



Now we can interpret the DFT of the $x(t) = \Pi(\frac{t}{4})$. The sampling interval is Δt = 1 second, and the obsesrvation length is T=20 seconds, and we have N=20 samples.

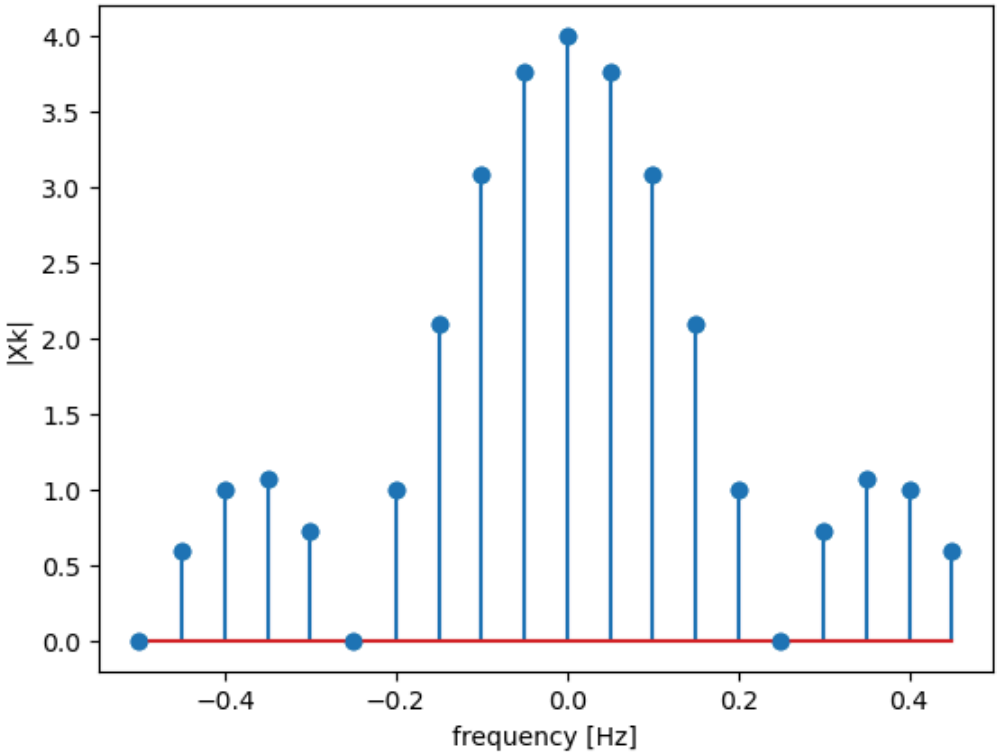
- We can recognize a bit of a sinc function.
- The DFT is computed from 0 Hz, for positive frequencies up to $\frac{fs}{2} = 0.5$ Hz, after which the negative frequencies follow from -0.5 to -0.05 Hz.
- $X(f) = 4\text{sinc}(4f)$ has its first null at $f = 0.25$Hz.

Task 0.4: Create symmetric plot

For convenient visualization, we may want to explicitly shift the negative frequencies to the left-hand side to create a symmetric plot. We can use `numpy.fft.fftshift` to do that. In other words, the zero-frequency component appears in the center of the spectrum. Now, it nicely shows a symmetric sprectrum. It well resembles the sinc-function (taking into account that we plot the modulus/absolute value). The output of the DFT still consists of N = 20 elements. To enable this, we set up a new frequency array (in Hz) on the interval $[-fs/2, fs/2)$.

Task 0.4: Read the code cell below before executing it and identify how the plot is modified based on the (new) specification of frequency. Note that it is more than just the `freq` variable!

In [5]:
```
abs_fft_shift = np.abs(np.fft.fftshift(np.fft.fft(xt)))
freq = np.arange(-0.5, 0.5, 0.05)
plt.stem(freq, abs_fft_shift)
plt.plot(freq, abs_fft_shift, 'o')
plt.ylabel('|Xk|')
plt.xlabel('frequency [Hz]');
```

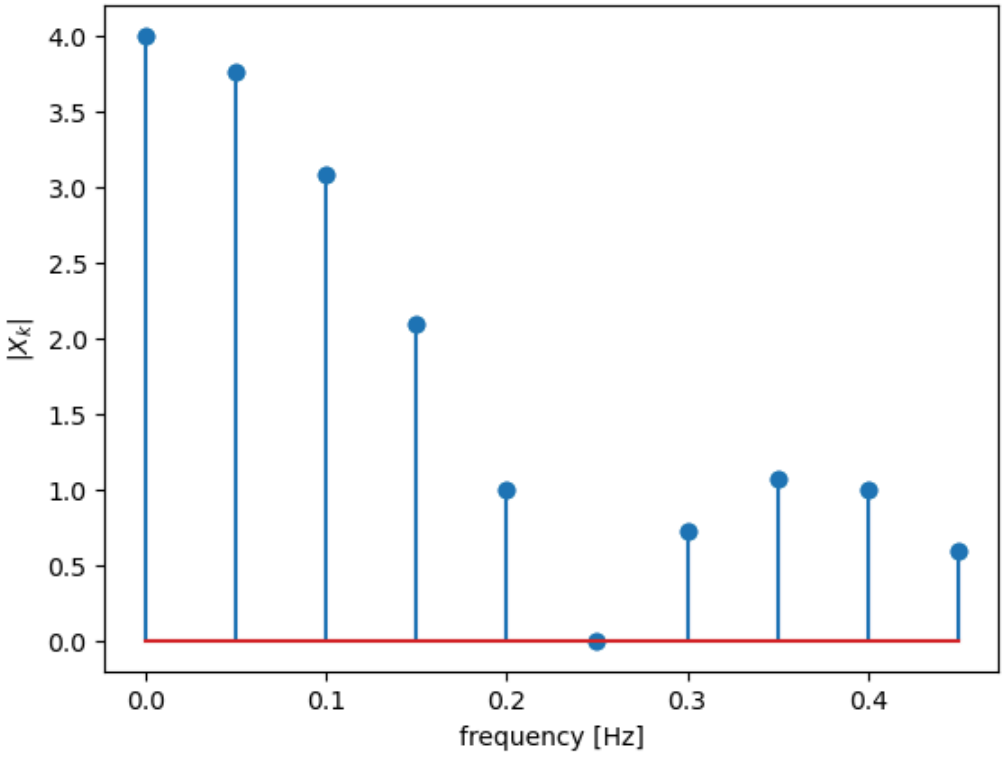


Task 0.5: Showing spectrum only for positive frequencies

In practice, because of the symmetry, one typically plots only the right-hand side of the (double-sided) spectrum, hence only the part for positive frequencies $f \geq 0$. This is simply a matter of preference, and a way to save some space.

Task 0.5: Can you identify what has changed (in the code and visually), compared to the previous plot?

In [6]:
```
N=len(xt)
abs_fft = np.abs(np.fft.fft(xt))
freq = np.arange(0.0, 1.0, 0.05)
plt.plot(freq[:int(N/2)], abs_fft[:int(N/2)], 'o')
plt.stem(freq[:int(N/2)], abs_fft[:int(N/2)])
plt.ylabel('$|X_k|$')
plt.xlabel('frequency [Hz]');
```



Task 0.6: Confirm that you understand how we have arrived at the plot above, which illustrates the magnitude (amplitude) spectrum for frequencies $f \in [0, f_s/2)$, rather than $[0, f_s)$.

Task 1: Application of DFT using simple cosine

It is always a good idea, in spectral analysis, to run a test with a very simple, basic signal. In this way you can test and verify your coding and interpretation of the results.

Our basic signal is just a plain cosine. We take the amplitude equal to one, and zero initial phase, so the signal reads $x(t) = \cos(2\pi f_c t)$, with f_c = 3 Hz in this exercise. With such a simple signal, we know in advance how the spectrum should look like. Namely just a spike at f = 3 Hz, and also one at f = -3 Hz, as we're, for mathematical convenience, working with double sided spectra. The spectrum should be zero at all other frequencies.

As a side note: the cosine is strictly a periodic function, not a-periodic (as above); still, the Fourier transform of the cosine is defined as two Dirac delta pulses or peaks (at 3 Hz and -3 Hz). You may want to check out the second worked example in Chapter 3 on the Fourier transform: Fourier transform in the limit.

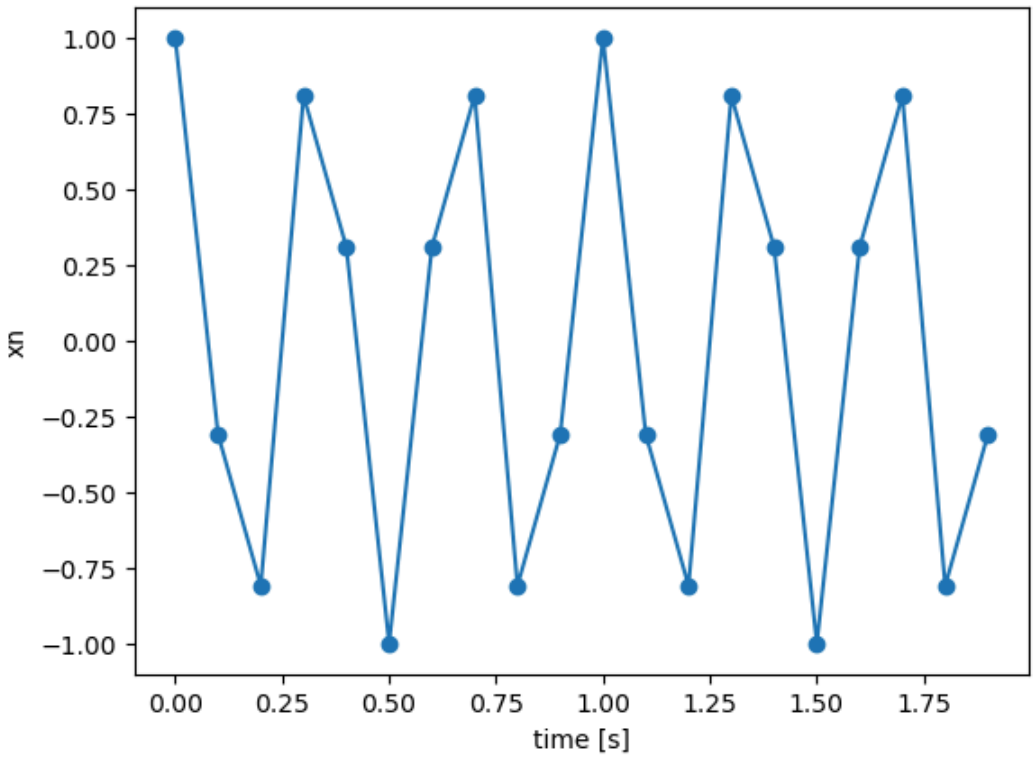
Task 1:

Create a sampled (discrete time) cosine signal by sampling at f_s = 10 Hz, for a duration of T = 2 seconds (make sure you use exactly N = 20 samples). Plot the sampled signal, compute its DFT and plot its magnitude spectrum $|X_k|$ with proper labeling of the axes (just like we did in the first part of this notebook). Include a plot of the spectrum of the sampled cosine signal using only the positive frequencies (up to $f_s/2$, as in the last plot of the previous task).

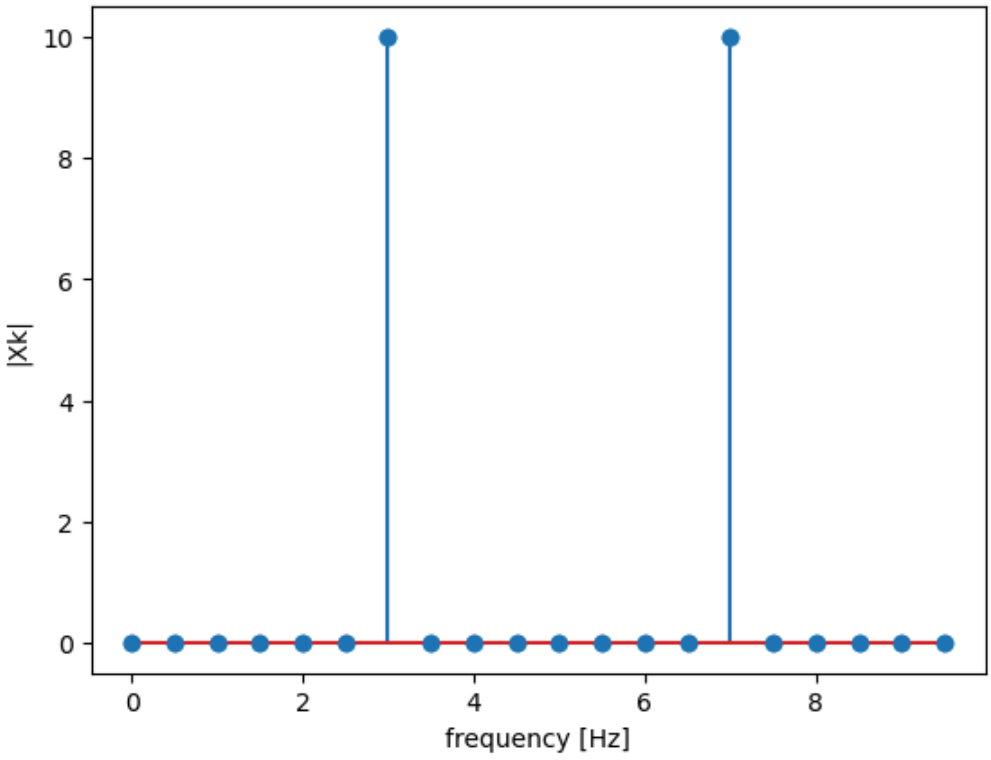
Note: you are expected to produce three separate plots.

In [13]:
```
### SOLUTION

fc=3
fs=10
```

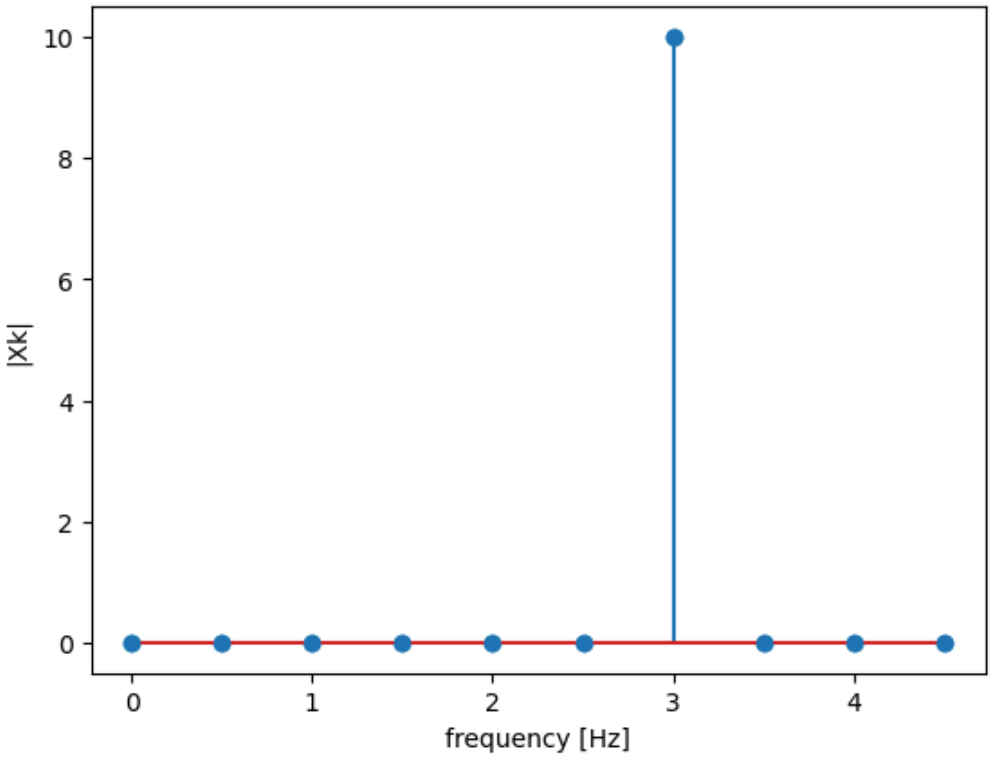
```
dt=1/fs
T=2
N=T*fs
t=np.arange(0,T,dt)
xt=np.cos(2*np.pi*fc*t)

plt.plot(t,xt, marker = 'o')

plt.xlabel('time [s]')
plt.ylabel('xn');
```



In [15]:
```
### SOLUTION

abs_fft = np.abs(np.fft.fft(xt))
freq=np.arange(0,fs,1/T)
plt.stem(freq, abs_fft)
plt.plot(freq, abs_fft, 'o')
plt.ylabel('|Xk|')
plt.xlabel('frequency [Hz]');
```



In [9]:
```
### SOLUTION

abs_fft = np.abs(np.fft.fft(xt))
freq=np.arange(0,fs,1/T)
plt.stem(freq[:int(N/2)], abs_fft[:int(N/2)])
plt.plot(freq[:int(N/2)], abs_fft[:int(N/2)], 'o')
plt.ylabel('|Xk|')
plt.xlabel('frequency [Hz]');
```



End of notebook.
