We wish to approximate an integral of the form

$$\mathcal{F}(\nu) = \int_0^\infty dt e^{i\nu t} f(t) \tag{1}$$

for a function $f(t)$ decays to zero over a timescale $\tau$. Taking $N$ samples $f_n = f(t_n)$ at equally spaced times $t_n = 0, T/N, 2T/N, \ldots (N-1)T/N$ where $T > \tau$,

$$\begin{aligned}
\mathcal{F}(\nu) &\approx \int_0^T dt e^{i\nu t} f(t) && (2)\\
&\approx \sum_{n=0}^{N-1} e^{i\nu t_n} f_n \Delta t \qquad (\Delta t = T/N) && (3)\\
&= \Delta t \sum_{n=0}^{N-1} e^{i\nu n T/N} f_n && (4)
\end{aligned}$$

This should be compared to the definition of the FFT, as implemented in `numpy`:

$$F_k := \sum_{n=0}^{N-1} f_n e^{-2\pi i n k/N}, \quad k = 0, \ldots, N-1 \tag{5}$$

We see that

$$\mathcal{F}(-2\pi k/T) = F_k \cdot \Delta t, \quad k = 0, \ldots, N-1 \tag{6}$$

i.e. evaluating $F_k \cdot \Delta t$ ($F_k$ the $k^{\text{th}}$ Fourier component) gives the required function, but at frequencies

$$\nu = -\frac{2\pi}{T} k = -\frac{2\pi}{N\Delta t} k \tag{7}$$

By periodicity under $k \to k + N$, $k$ can range over any $N$ consecutive integers, however by default `np.fft.fft` provides the output in a particular 'standard' order. Referring to a range of $N$ integers centered on zero, this is: 'zero frequency, positive frequencies, negative frequencies' i.e. $k = 0$, $k = 1, 2, \ldots, (N-1)/2$, $k = -(N-1)/2, \ldots -1$ for $N$ odd.[1] Applying `np.fft.fftshift()` to the result of `np.fft.fft` moves the elements into the more sane order of strictly increasing frequencies.

You can construct the frequencies directly using (7) or via `np.fft.fftshift(np.fft.fftfreq(N, d=dt))` which returns the frequencies (ascending order) with spacing $1/(N\Delta t)$ so that only multiplication by $-2\pi$ is required. Note the maximum frequency scales as $1/\Delta t$ i.e. higher resolution in real real space results in a larger range of frequencies in reciprocal space,[2] whilst the frequency resolution scales as $1/T$ i.e. is set by the sample time.

**Using ifft** To avoid the complication of having to negate the frequencies, it is simpler to use the inverse Fourier transform provided by numpy with the option `norm='forward'`, which gives

$$F_k = \sum_{n=0}^{N-1} f_n e^{2\pi i n k/N} \tag{8}$$

so that out frequencies are in the right order. **In summary, use the code**

[1] For $N$ even it goes $k = 0$, $k = 1, 2, \ldots, N/2-1$, $k = -N/2, -(N-2)/2, \ldots, -1$ i.e. the negative range gets the $-N/2$ value (Nuquist frequency). A subsequent `fftshift` would leave you with $k = -N/2, -N/2+1, \ldots, N/2-1$.

[2] Although this does not necessarily provide additional information; beyond a certain frequency (determined by the most rapidly changing part of $f$) the spectrum will be zero.

```
# times - array of equally spaced times at which function was sampled
dt = times[1] - times[0]
N = len(sample) # for higher frequency resolution, consider oversampling
sample_fft = np.fft.fftshift(np.fft.ifft(sample, norm='forward', n=N))
nus = 2 * np.pi * np.fft.fftshift(np.fft.fftfreq(N), d=dt)
```

Note the asymmetry of the frequency spectrum when $N$ is even (see footnote 1) mean the output here will start from $\nu = -(2\pi/T)(N/2)$ compared to $\nu = -(2\pi/T)((N-1)/2)$ if one was using the corresponding call to `fft`:

```
sample_fft = np.flip(np.fft.fftshift(np.fft.fft(sample, n=N)))
nus = np.flip(- 2 * np.pi * np.fft.fftshift(np.fft.fftfreq(N), d=dt))
```

by virtue of `np.flip`, which was required to put the negated frequencies `nus` into ascending order and move the corresponding elements of `sameple_fft` in the same way.

**Zero-padding** For a function known to have decayed to 0 by the final sample time i.e. $f(t) = 0\ \forall t \geq T$, you may consider appending zeros to the array $\{f_n\}_n^N$ of samples in order to increase the resolution in frequency. As commented above, the `n` parameter of `np.fft.ifft` can be used for this purpose: setting $n = M$ for $M > N$ results in the sample being padded with $M - N$ zeros (for $M < N$ it is cropped). Coincidently, the FFT algorithm is most efficient when the number of samples is a power of 2, so you may want to consider setting

```
M=2**(int(np.ceil(np.log2( FAC * N ))))
```

where `FAC` is the desired oversample factor.

**Endpoint correction** Computing integrals in this way is susceptible to truncation error, in particular at large positive or negative $\nu$. This is discussed in Ref. [1, Sec. 13.9]. An endpoint correction scheme based on this text is implemented in `improved_fft.py`.

## References

[1] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3e. Cambridge University Press (Cambridge, UK, 2007).