Intro to Python II

ASTR 2910 ⭑ Week 5

Recap of last time

Variables: Storing information for later use; multiple **data types** available

Boolean operators: Making comparisons between quantities

Conditionals: Branching decisions (based on Boolean expressions)

Loops: Repetitive actions (over an iterable or until a condition is met)

Functions

Introduction to functional programming

Functional programming is the practice of writing modular pieces of code, called functions, that are designed to perform generic tasks on inputs of the same kind.

Why use it?

- Shorter (“cleaner”) code
- Easier to test (because each part can be tested alone)
- Avoid errors from copy-pasting code but forgetting to update each copy

We’ve already encountered several functions! `print()`, `.append()`, etc.

Structure of a Python function

Special Python keyword that starts every function definition

Name for your function (same rules as for variable names)

Parentheses indicate what arguments a function takes (here, none)

```python
def my_function():
    pass
```

Colon and indentation tell Python what code to include in the function

Function body:
Can contain any valid Python code of any length;
`pass` keyword allows body to be empty

Using a Python function

When you run your code, Python will *read and store* your function definitions, but it won't actually *run* the code inside of them.

To use a function, you have to *call* it (**after** the definition):

```
def my_function():
    print('Hello, world!')

my_function()
```

Function calls have two parts:

1. Name of the function
2. Arguments, enclosed with ()

Empty parentheses = no arguments

Arguments

The power of functions comes from their ability to take inputs (called *arguments*).

```
def random_calculation(a, b, c, d):
    calculate = c*(a+b) - d
    print(calculate)
```

In a *definition*, list a placeholder name for each argument in the (). You can have as many as you need.

```
num = 2
random_calculation(1, num, 3, 4)
```

In a *call*, provide one value for each argument, in order. You can pass values or variables (even if they have different names).

Scope

Global scope: Definitions that your entire code has access to

Local scope: Definitions that only the function has access to (includes anything created within the body + any argument names)

Definitions in a local scope will cease to exist when the function call is over!

```
def random_calculation(a, b, c, d):
    calculate = c*(a+b) - d

random_calculation(1, 2, 3, 4)
print(calculate)
```

```
extra_string = '!'
def my_function(string_to_print):
    print(string_to_print + extra_string)

my_function('Hello, world!')
```

`calculate` doesn't exist outside of the function

`extra_string` can be accessed inside functions

Scope

Good practice: Passing in everything your function needs to run

Bad practice:

```
string_to_print = 'Hello, world!'
def my_function():
    print(string_to_print)
```

(Rare) exceptions: Constants (though you can often get these from packages)

`return`

We can print out calculations that we do inside functions, but how do we store those results for later use?

```
def random_calculation(a, b, c, d):
    calculate = c*(a+b) - d
    return calculate

result = random_calculation(1, 2, 3, 4)
print(result)
```

Syntax for storing function results in a variable

Functions with no explicit `return` will still return `None`.

Code after a return statement will be ignored when a function is called!

Advanced options

You can call functions within other functions, or chain functions together.

Optional arguments and default values:

```
#Function to model the galaxy background with a 2D median filter and subtract from the image
#The default filter size was chosen through trial and error
def sub_galaxy_background(hdul, fn_out, medfilt_size=41, verbose=False):
```

Returning multiple things:

```
def return_multiple_things(a, b):
    sum2 = a + b
    diff2 = a - b
    product2 = a * b
    return sum2, diff2, product2

s, d, p = return_multiple_things(2, 4)
```

These are both tuples! Breaking them down like this is called *unpacking*.

Best practices

1. Writing *docstrings* to describe the arguments/parameters and expected behavior of your function

Proper

… and overkill in most cases

```
def load_directory_images(path):
    '''
    Loads a directory's worth of images into convenient
    storage units. Requires astropy.io.fits.
    Note: All Images in directory must be of same shape.

    Parameters
    ----------
    path: str
        path to the directory you wish to load, as a string.

    Returns
    -------
    image_stack: array_like
        A stack of all images contained in the directory.
        Array of shape (N,X,Y) where N is the number of images,
        and X,Y are the dimensions of each image.
    image_dict: dict
        A dictionary containing headers for each image, the keys
        are the same as the indices of the corresponding
        image in the image_stack
    '''
```

For personal use

Could even add more detail here

```
def sub_galaxy_background(hdul, fn_out, medfilt_size=41, verbose=False):
    '''
    hdul: a Condor image hdul (4 extensions: empty, data, sigma image, mask)
    '''
```

Best practices

2. Validating input

```
def return_multiple_things(a, b):
    assert (type(a) == int or type(a) == float) and (type(b) == int or type(b) == float), 'a and b must be numbers'
    sum2 = a + b
    diff2 = a - b
    product2 = a * b
    return sum2, diff2, product2
```

Syntax: `assert` [Boolean expression], "string to be printed if assertion fails"

Also overkill for most cases when your code will just be used by you, but you should use assertions as needed to check yourself.

Exercise

In astronomy, the flux **F** received from a star is related to its intrinsic brightness (luminosity) **L** and the distance **d** to the star by the following equation:

$$F = \frac{L}{4\pi d^2}$$

Write a function called `calculate_flux` that takes **L** and **d** as arguments and returns **F**. Use your function and the below values (no conversions necessary) to calculate the flux of the Sun at the Earth.

$L_{Sun} = 3.828 * 10^{26}$ W

$d_{Sun} = 1.496 * 10^{11}$ m

Packages

What are packages?

Quite literally: “Packages” of code (functions, constants, etc) written by others that extend the functionality of Python.

`conda` is one of the most popular *package managers*: software that provides an easy, centralized way to install and manage packages.

- `conda` allows you to create distinct environments with different sets of packages (the default environment is `base`)
    - *In VS Code: View > Command Palette > “Python: Select Interpreter”*
- Packages can be installed in an environment with `conda` or `pip`

Anyone can write a package (even you; check out Code/Astro!), and you can package code even if you don’t publish it.

How to use packages

The most famous line of code in all of astronomy:

```
import numpy as np
```

Syntax: `import` [package name] (`as` [short name])

Functions can then be accessed with “dot notation”: `np.function()`

Importing specific functions:

```
from numpy import arange, linspace
```

Importing everything from a package:

```
from numpy import *
```

In both cases, functions are then accessed by their names alone (no dot notation).

How to use packages

Import statements are usually collected at the top of a script or notebook:

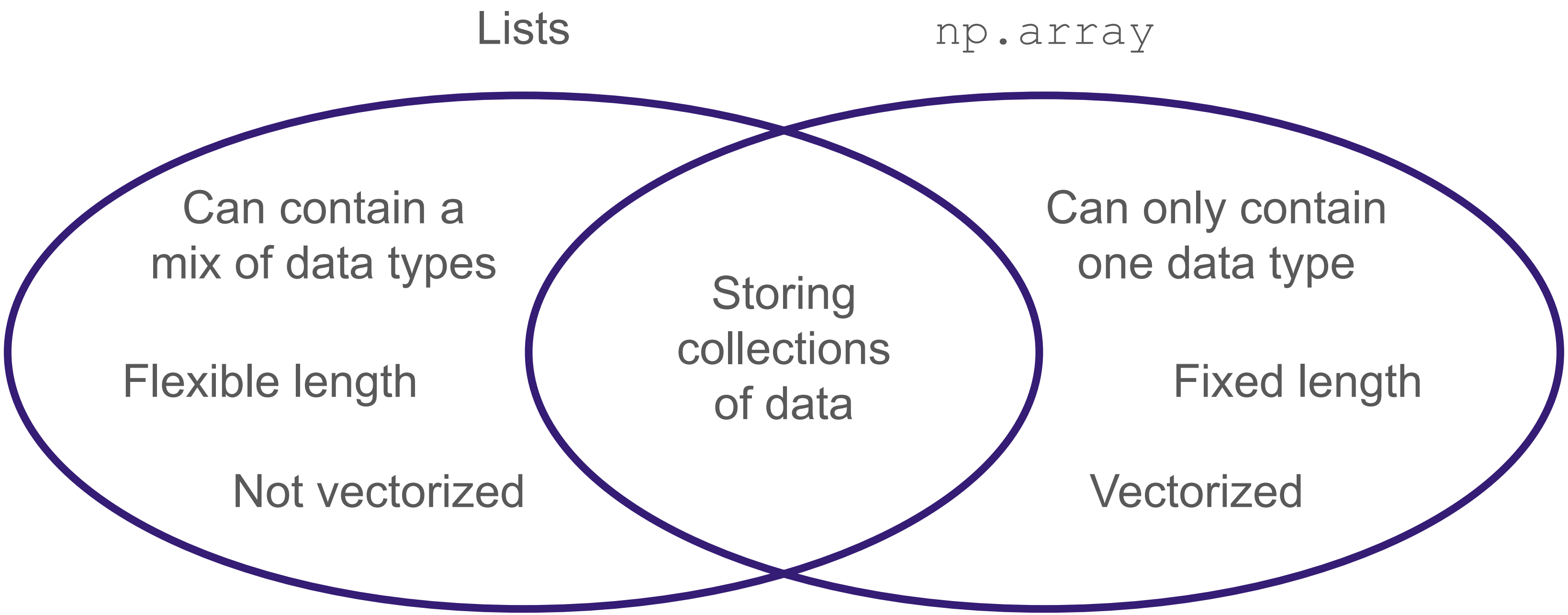
```
#Necessary imports and packages
import glob
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from tqdm.notebook import tqdm
from astromy_ds9 import ds9_norm
from scipy.optimize import curve_fit

from astropy.io import fits
from astropy.coordinates import SkyCoord, match_coordinates_sky
import astropy.units as u
from astropy.wcs import WCS
from astropy.table import QTable, MaskedColumn
from astropy.convolution import convolve_fft
from astropy.stats import SigmaClip, mad_std, sigma_clipped_stats, gaussian_sigma_to_fwhm, gaussian_fwhm_to_sigma

from photutils.detection import DAOStarFinder
from photutils.aperture import CircularAperture, ApertureStats, CircularAnnulus
from photutils.psf import PSFPhotometry, IntegratedGaussianPRF, SourceGrouper
from photutils.psf.matching import create_matching_kernel
from photutils.background import SExtractorBackground, Background2D, LocalBackground
from photutils.utils import make_random_cmap
```

numpy

Defines the (very useful) array data type, and contains a bunch of useful functions for working with arrays and doing math/stats.



`numpy`

Arrays are created simply by using `np.array()` on an existing list. Or you can generate an array with a certain shape and replace the values one by one.

Arrays can be multi-dimensional (think of a 2D image).

Useful array attributes:

- `shape`: Tuple describing the length of each axis of the array
- `size`: The number of values in the array
- `dtype`: The type of data stored in the array
- `T`: The transpose of the array

`numpy`

Indexing and slicing 1D arrays is done identically to lists. For multi-dimensional arrays, indexing can take two forms:

`arr_2d[1][2]` == `arr_2d[1,2]`

Slicing is a bit more complicated, as each dimension can be accessed independently. For a 10x10 2D array, for example:

- `arr_2d[0, :]` will select the entire first row
- `arr_2d[0]` will also select the entire first row
- `arr_2d[:, 0]` will select the entire first column (the second dimension)

In general, commas separate the `start:stop:step` for each dimension.

Useful `numpy` functions

Generating data: `np.zeros()`, `np.arange()`, `np.linspace()`

Statistical distributions: `np.random.normal()`, `np.random.uniform()`

Math: `np.sin()`, `np.tan()`, `np.exp()`, `np.mean()`, `np.std()`, `np.sum()`

Manipulating arrays: `np.vstack()`, `np.hstack()`, `np.flatten()`

Boolean operations on an array: `np.where()`

NaN: `np.nan`

Everything you could possibly want to know is described in the documentation. I usually get there by Googling “numpy [function_name]”.

Exercise

Use `np.random.normal` to generate 100 random numbers drawn from a normal distribution, with a mean and standard deviation of your choice.

Then, calculate the mean and standard deviation of the resulting array, print the results, and compare them to your inputted values.

scipy

Routines for more complex tasks with scientific applications:

- Numerical integration and differentiation
- Optimization algorithms
- Linear algebra operations (like quick matrix inversion)
- Signal and image processing

(For more information: Chapter 7 in the textbook + documentation + this class)

`astropy`

Actively developed/maintained by practicing astronomers! Useful bits include:

- `astropy.units` subpackage, which provides a convenient way to associate physical units with quantities (and convert between them)
- `astropy.coordinates` subpackage for handling sky coordinates
- Support for reading in astronomy-specific data formats (FITS files)

(For more: Chapter 8 in the textbook + documentation + this class)