Review

SURP 2022 Python Bootcamp

Ohio State Astronomy

Slides by: James W. Johnson

Objectives

- Built-in Data Types: *int*, *float*, *list*, *tuple*, *dict*, *array*, etc.
- Conditionals: *if/else* statements
- Loops: *for* and *while*
- Functions: *def* statements
- Catching Exceptions: *try*/*except*/*finally*

Numeric Types

Built-in numeric types are *int* (integers) and *float* (real numbers)

- Some packages implement their own numeric types (e.g. NumPy: numpy.int64)

Numeric operations

=, ==, +, -, *, /, //, %, +=, -=, *=, /=, //=, %=

Recall: x += y is the same as x = x + y

/ vs. //: true division vs. floor division (i.e. integer quotient)

- 3 / 2 returns 1.5, but 3 // 2 returns 1

%: modulo (calculates remainder)

- 5 % 2 returns 1; 5 % 3 returns 2; 5 % 1 returns 0

Strings

Text - declared with either single ‘’ or double “” quotes

Triple quotes declare a multi-line string (This will come up again later)

Are compatible with the += operator

In Python 3, strings are *unicode* by default, meaning you can use special characters without changing anything

```
In [1]: x = "This is a "

In [2]: x += "test string."

In [3]: x
Out[3]: 'This is a test string.'
```

The Print Function

In C/C++ print statements are done with *printf* in stdio.h, and *cout* in iostream

Python: *print(<things to print>)*

- Can pass multiple parameters separated by commas, and they'll print with spaces between them

Parentheses are required in Python 3, but not Python 2 (which is deprecated so you shouldn't be using it anyway)

Type Casting

Python objects can be converted between compatible types

Strings containing numbers can be converted to numeric types

Integers can be converted to floats and vice versa

```
In [1]: x = "3"

In [2]: int(x)
Out[2]: 3

In [3]: float(x)
Out[3]: 3.0

In [4]: x = 3.1

In [5]: int(x)
Out[5]: 3

In [6]: float(3)
Out[6]: 3.0

In [7]: 3.
Out[7]: 3.0
```

Import Statements

Used to load a python library into your current code

Analogous to *#include* and *using* in C/C++

Can assign imported modules new names upon import with *from* and *as*

from ___ *import* * imports all contents

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

In [2]: import numpy as np

In [3]: from math import log10 as log

In [4]: import sys

In [5]: log(10)
Out[5]: 1.0

In [6]: log(3)
Out[6]: 0.47712125471966244

In [7]: plt.figure()
Out[7]: <Figure size 640x480 with 0 Axes>
```

List

A sequence of objects

- Created with square brackets [ ]
- Can be any mix of types
- Access an element of the sequence by its position
  - Negative indices start at the *end* of the list and count backwards
  - Modification proceeds in the same manner
- Append function adds an element to the end
  - Can also use '+' to combine lists

```
In [1]: example = [1, 2, "string", 3]

In [2]: example
Out[2]: [1, 2, 'string', 3]

In [3]: example[0]
Out[3]: 1

In [4]: example[2]
Out[4]: 'string'

In [5]: example.append("appended")

In [6]: example
Out[6]: [1, 2, 'string', 3, 'appended']

In [7]: example[-1]
Out[7]: 'appended'
```

Tuple

An *immutable* sequence of objects

- Created with parentheses ( )
- Can be any mix of types
- Access an element of the sequence by its position
  - Negative indices start at the *end* of the tuple and count backwards
  - Modification *not allowed* (immutability)
  - Can still use '+' to add elements to the end

```
In [1]: example = (1, 2, "string", 3)

In [2]: example
Out[2]: (1, 2, 'string', 3)

In [3]: example[0]
Out[3]: 1

In [4]: example[2]
Out[4]: 'string'

In [5]: example.append("appended")
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-7775d98792ef> in <module>
----> 1 example.append("appended")

AttributeError: 'tuple' object has no attribute 'append'

In [6]: example[-1]
Out[6]: 3

In [7]: example[1] = 1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-d244a90d0c3c> in <module>
----> 1 example[1] = 1

TypeError: 'tuple' object does not support item assignment

In [8]: example += (4, 5)

In [9]: example
Out[9]: (1, 2, 'string', 3, 4, 5)
```

Set

Ensure uniqueness

- *list(set(some_list))* will remove duplicate elements
- Created with the *set()* function or with {} enclosing elements (be careful w/this, see next slide)
- Don't allow indexing, so aren't as commonly used as lists, tuples, and dictionaries
- Have some other useful function such as union and intersection ('|' and '&')

```
In [1]: x = list(range(10))

In [2]: x
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: x[3] = 4

In [4]: x
Out[4]: [0, 1, 2, 4, 4, 5, 6, 7, 8, 9]

In [5]: list(set(x))
Out[5]: [0, 1, 2, 4, 5, 6, 7, 8, 9]
```

Dictionary

Python's version of a *hash table*.

- Can be used to map objects to other objects
- Created with {}
- Stored values can be accessed via their *key*
- *keys()* function returns each key
- Popular method of storing data b/c keys can be strings which describe the data, allowing very readable code

```
In [1]: example = {"a": 1, "b": 2, 3: "c"}

In [2]: example["a"]
Out[2]: 1

In [3]: example[3]
Out[3]: 'c'

In [4]: example.keys()
Out[4]: dict_keys(['a', 'b', 3])

In [5]: example["mass"] = list(range(10))

In [6]: example
Out[6]: {'a': 1, 'b': 2, 3: 'c', 'mass': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
```

Which Data Type Should I Use?

Do you need a logical *key-value* connection?

If yes: use a dictionary

Do you need to ensure uniqueness of each element, or perhaps union or intersection operations?

If yes: use a set

Do you need to ensure that the contents will never change?

If yes: use a tuple

If you answered no to all of these, a list should suffice.

Arrays

In practice the same as a list, but has some special implementation of tracking data types under the hood, and allow some calculations to be automatically “vectorized”

- Can often speed up code
- There is a built-in array object, but in practice, most people use the NumPy array

The single most important thing to remember about arrays:

LISTS AND ARRAYS ARE ***NOT*** THE SAME THING

Lists vs. Arrays

They are ***different objects*** meant to store similar data

Example: NumPy arrays allow multiplication with another array. A list does not.

Advice: Pick one of either lists or arrays for a given program and stick to it

```
In [1]: import numpy as np

In [2]: x = list(range(10))

In [3]: y = np.array(range(10))

In [4]: x
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [5]: y
Out[5]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [6]: type(x)
Out[6]: list

In [7]: type(y)
Out[7]: numpy.ndarray

In [8]: y * y
Out[8]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

In [9]:  x * x
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-40f972982daf> in <module>
----> 1 x * x

TypeError: can't multiply sequence by non-int of type 'list'
```

Lists vs. Arrays

They are ***different objects*** meant to store similar data

It's probably safe to say that a large portion of all modern scientific code is written using NumPy arrays

```
In [1]: import numpy as np

In [2]: x = list(range(10))

In [3]: y = np.array(range(10))

In [4]: x
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [5]: y
Out[5]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [6]: type(x)
Out[6]: list

In [7]: type(y)
Out[7]: numpy.ndarray

In [8]: y * y
Out[8]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

In [9]:  x * x
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-40f972982daf> in <module>
----> 1 x * x

TypeError: can't multiply sequence by non-int of type 'list'
```

Slicing

A technique used to pull multiple items from array-like objects

- General rule: *start:stop:stepsize*
- *Start*, *stop*, and *stepsize* can be omitted, and Python defaults to the beginning or end with a step size of 1, depending on what's omitted

Achieved by separating indices with a colon

Negative step sizes go through the array in reverse order

```
In [1]: x = list(range(10))

In [2]: x
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: x[2:5]
Out[3]: [2, 3, 4]

In [4]: x[3:9]
Out[4]: [3, 4, 5, 6, 7, 8]

In [5]: x[1:-1]
Out[5]: [1, 2, 3, 4, 5, 6, 7, 8]

In [6]: x[:-1]
Out[6]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

In [7]: x[1:]
Out[7]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [8]: x[1::2]
Out[8]: [1, 3, 5, 7, 9]

In [9]: x[::-1]
Out[9]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Conditions

Conduct different operations based on whether or not a given condition is satisfied

Uses boolean logic: *True* and *False*

- Note: Numbers can be used as well – anything nonzero evaluates to True.

Tip: Don't compare to *True* and *False*

- *if x == True* and *if x == False* are simply *if x* and *if not x*

```
In [1]: x = 1

In [2]: if x > 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is positive.

In [3]: x = 0

In [4]: if x > 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is zero.

In [5]: x = -1

In [6]: if x > 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is negative.
```

Conditions

Conduct different operations based on whether or not a given condition is satisfied

Uses boolean logic: *True* and *False*

- Note: Numbers can be used as well – anything nonzero evaluates to True.

Note: *if*/*elif*/*else* blocks executed *in order*.

```
In [1]: x = 1

In [2]: if x >= 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is positive.

In [3]: x = 0

In [4]: if x >= 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is positive.

In [5]: x = -1

In [6]: if x >= 0:
   ...:     print("x is positive.")
   ...: elif x == 0:
   ...:     print("x is zero.")
   ...: else:
   ...:     print("x is negative.")
   ...:
x is negative.
```

Loops

Two types

- For- and while-loops

Both execute the same block of code some number of times

Both types of loops can be forced to terminate with the command “break”, and to start the next iteration with “continue”

For-loops

Should be used when you know *exactly* how many times the block should repeat

Come in two flavors:

- *Explicit* for loop: "*for i in <some iterable>*" followed by an indented block (example: top)
- *Implicit* for loop: occurs within a list comprehension (example: bottom)
  - Note that list comprehensions return a *list*, so you must type-cast to an array if you want to write your program using arrays

```
In [1]: for i in range(10):
   ...:     if i == 5: continue
   ...:     if i == 8: break
   ...:     print(i**2)
   ...:
   ...:
0
1
4
9
16
36
49

In [2]: [i**2 for i in range(10)]
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

While-loops

Should be used when you *don't* know how many times the block should repeat

- You should break this rule when the alternate version is significantly more readable (this is quite universal advice)

while True: <...> break is not uncommon, but considered bad practice by some

- Advantage: The loop will *always* execute at least once. Other languages achieve this with what is called a *do-while* loop.

```
In [1]: import numpy as np

In [2]: x = np.random.normal()

In [3]: x
Out[3]: -1.2309450778924569

In [4]: while x < 0:
   ...:     print(x)
   ...:     x = np.random.normal()
   ...:
-1.2309450778924569
-0.9441969498633814

In [5]: x
Out[5]: 0.599929527694834

In [6]: while x > 0:
   ...:     print(x)
   ...:     x = np.random.normal()
   ...:
0.599929527694834
0.5108733408026233
0.012998073685294384
1.1176695275075066
1.19053299714839

In [7]: x = 0

In [8]: while True:
   ...:     x += 1
   ...:     if x > 10:
   ...:         break
   ...:

In [9]: x
Out[9]: 11
```

Functions

Often referred to as “methods” in other languages

Created with the *def* keyword followed by an indented block. Between the *def* statement and the body of the function is where you should put a *docstring*.

Astronomers (and scientists in general) are *notorious* for thin documentation if they document at all.

```
In [1]: def f(x):
   ...:     """
   ...:     Calculate the value of x squared.
   ...:
   ...:     Parameters
   ...:     ----------
   ...:     x : real number
   ...:         The number to square
   ...:
   ...:     Returns
   ...:     -------
   ...:     y : real number
   ...:         The value of x squared.
   ...:     """
   ...:     return x**2
   ...:

In [2]: f(2)
Out[2]: 4

In [3]: f(13)
Out[3]: 169

In [4]: f(7.4)
Out[4]: 54.760000000000005
```

Functions: The Implicit Return

Unless otherwise specified, a function will return *None*.

In order to obtain an object from a function, you have to override this with a *return* statement

A note about variable scope: variables declared inside a function cannot be accessed outside the function.

```
In [1]: def f(x):
   ...:     """
   ...:     Calculate the value of x squared
   ...:     """
   ...:     print(x**2)
   ...:     # return (or: return None)
   ...:

In [2]: x = f(2)
4

In [3]: x is None
Out[3]: True

In [4]: def f(x):
   ...:     """
   ...:     Calculate the value of x squared
   ...:     """
   ...:     return x**2
   ...:

In [5]: x = f(2)

In [6]: x is None
Out[6]: False

In [7]: x
Out[7]: 4
```

Functions: An Alternative for One-Liners

One line function: *lambda*

By nature don't have any error-handling or documentation attached to them, so should only be used when this isn't necessary – a *lambda* worth documenting is better replaced by a *def* with a one-line *return* and a docstring

```
In [1]: f = lambda x: x**2

In [2]: f(3)
Out[2]: 9

In [3]: f(4)
Out[3]: 16

In [4]: g = lambda x: x.split()

In [5]: g("An example string")
Out[5]: ['An', 'example', 'string']
```

Catching Exceptions: try-except

Catch exceptions before their raised and handle them.

Different types of exceptions can be treated differently by specifying them in the except statement.

The exception can be raised “as is” with the python keyword *raise*

```
In [1]: import math as m

In [2]: try:
   ...:     print(m.log10(-1))
   ...: except ValueError:
   ...:     print("Caught ValueError!")
   ...: except:
   ...:     print("Caught some different error!")
   ...:
Caught ValueError!

In [3]: try:
   ...:     print(m.log10("0"))
   ...: except ValueError:
   ...:     print("Caught ValueError!")
   ...: except:
   ...:     print("Caught some different error!")
   ...:
Caught some different error!
```

Catching Exceptions: try-except-finally

Use a *finally* block when you need something to *always run* regardless of the errors that may or may not be raised.

Example: freeing up memory

Disclaimer – return statements in a *finally* block will always override previous return statements in the try-except block

```
In [1]: import math as m

In [2]: try:
   ...:     print(m.log10(-1))
   ...: except ValueError:
   ...:     print("Caught ValueError!")
   ...: except:
   ...:     print("Caught some different error!")
   ...: finally:
   ...:     print("This will always run.")
   ...:
Caught ValueError!
This will always run.
```

Pattern-Matching: Python 3.10

Python 3.10 (released October 4, 2021) introduced the new *match-case* syntax

- Similar to the *switch-case* construction in C/C++
- Usually implicitly includes some form of type-checking and length-matching for short arrays

Case statements can introduce new local variables

_ represents some default scenario

```
In [1]: def f(point):
   ...:     match point:
   ...:         case [0, 0]:
   ...:             print("Origin")
   ...:         case [x, 0]:
   ...:             print("On the x-axis. x =", x)
   ...:         case [0, y]:
   ...:             print("On the y-axis. y =", y)
   ...:         case [x, y]:
   ...:             print("On neither axis. x =", x, "and y =", y)
   ...:         case _:
   ...:             raise TypeError("Not a 2-dimensional data point.")
   ...:

In [2]: f([0, 0])
Origin

In [3]: f((1, 0))
On the x-axis. x = 1

In [4]: f([0, 1])
On the y-axis. y = 1

In [5]: f((1, 1))
On neither axis. x = 1 and y = 1
```

Pattern-Matching: Python 3.10

Python 3.10 (released October 4, 2021) introduced the new *match-case* syntax

- Similar to the *switch-case* construction in C/C++
- Usually implicitly includes some form of type-checking and length-matching for short arrays

Logical “or” function can be used with | character…

```
In [1]: def f(point):
   ...:     match point:
   ...:         case [x, 0] | [0, x]:
   ...:             print("On an axis.")
   ...:         case _:
   ...:             print("Not on an axis.")
   ...:

In [2]: f([0, 0])
On an axis.

In [3]: f((1, 0))
On an axis.

In [4]: f([0, 1])
On an axis.

In [5]: f((1, 1))
Not on an axis.

In [6]: f((1, 1, 1))
Not on an axis.
```

Pattern-Matching: Python 3.10

Python 3.10 (released October 4, 2021) introduced the new *match-case* syntax

- Similar to the *switch-case* construction in C/C++
- Usually implicitly includes some form of type-checking and length-matching for short arrays

Logical "or" function can be used with | character…

… and can be used with *as* to capture the value as a local variable

```
In [1]: def f(cmd):
   ...:     match cmd:
   ...:         case ["walk", ("n" | "s" | "e" | "w") as direction]:
   ...:             directions = {
   ...:                 "n": "north",
   ...:                 "s": "south",
   ...:                 "e": "east",
   ...:                 "w": "west"
   ...:             }
   ...:             print("Walking", directions[direction], "....")
   ...:         case _:
   ...:             raise ValueError("Unrecognized command.")
   ...:

In [2]: f(["walk", "n"])
Walking north ....

In [3]: f(["walk", "s"])
Walking south ....

In [4]: f(["walk", "e"])
Walking east ....
```

Pattern-Matching: Python 3.10

Python 3.10 (released October 4, 2021) introduced the new *match-case* syntax

- Similar to the *switch-case* construction in C/C++
- Usually implicitly includes some form of type-checking and length-matching for short arrays

PEP 636 presents a comprehensive tutorial

- https://peps.python.org or google “pep 636”
- PEP: Python Enhancement Proposal

```
In [1]: def f(cmd):
   ...:     match cmd:
   ...:         case ["walk", ("n" | "s" | "e" | "w") as direction]:
   ...:             directions = {
   ...:                 "n": "north",
   ...:                 "s": "south",
   ...:                 "e": "east",
   ...:                 "w": "west"
   ...:             }
   ...:             print("Walking", directions[direction], "....")
   ...:         case _:
   ...:             raise ValueError("Unrecognized command.")
   ...:

In [2]: f(["walk", "n"])
Walking north ....

In [3]: f(["walk", "s"])
Walking south ....

In [4]: f(["walk", "e"])
Walking east ....
```

Useful Built-In Functions

any: Iterates over a list/array and determines if at least one element is *True*

all: Iterates over a list/array and determines if all elements are *True*

zip: Combine lists/arrays into a 2-D list/array component-wise

map: Iterate a function over an array(s) of values

filter: Remove elements from a list/array which don't meet specific criteria

min: Calculate minimum value of some set of numbers

max: Calculate maximum value of some set of numbers

In python 3, *zip*, *map*, and *filter* return a special type of object which needs to be cast to a list, tuple, etc.