



# Basic Python

Tips, tool, and techniques for those  
starting in Python

# Topics

Prelude

Data structures

Iterators

Functions

**Exercises**

Plotting

**Exercises**

Go to syzygy:

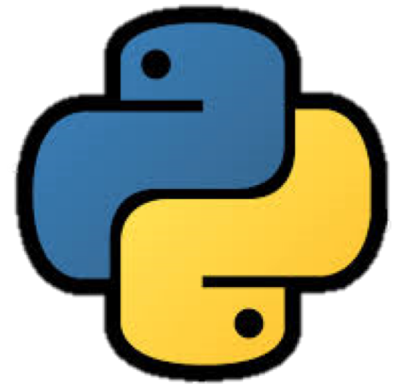
<https://uvic.syzygy.ca/>

Open a new terminal and clone this repository:

<https://bit.ly/2pp5kDx>

Topic 0

# Prelude



## Prelude

# What is Python?

- Invented in the early '90s by Guido Van Rossum
- **CPython Interpreter**
  - Reads your code 📖
  - Parses it 👁️
  - Checks for **syntax** errors ✓
  - Generates **instructions** 👉
  - Steps through instructions ▶▶



## Prelude

# Beyond CPython

There are other ways of running your Python code:



The best way to get CPython + Scientific libraries

# Prelude

## 2 vs. 3

Python 3 > Python 2

Stick to 3.5+



Python 3



Python 2

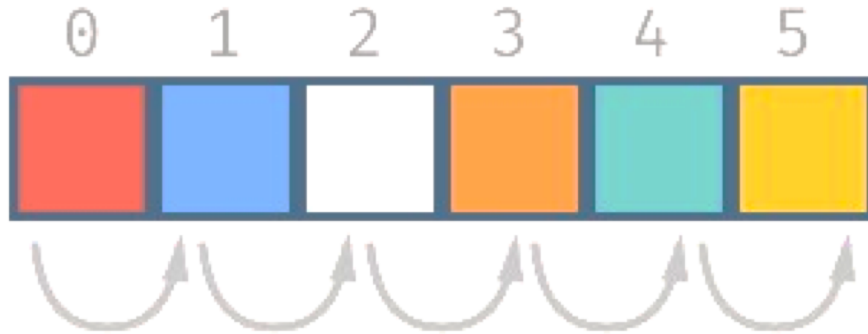


# Prelude

## Variables

```
3    foo = [3,1,2]
4    bar = sort(foo)
5    print(foo)
6    # [1,2,3]
```

What happened to foo?



Topic 1

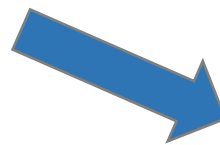
# Data Structures

## Topic 1

# Data Structures

### Lists

- Most of us are familiar
- Store data in **order**
- Can **loop** through
- Can look up by **index**
- Slow **searching**



```
abc = [1,2,3]  
print(abc[2])  # 3
```

# Data Structures

## Dictionaries

- Store data as key: value pairs
  - Keys can be:
    - Numbers
    - Strings
    - Tuples of numbers, strings
  - Values can be: anything!
  - “**Instant**” lookup by key!
    - $O(1)$  even if  $1e6$  entries!
  - Slower than lists for looping

# Data Structures

## Dictionaries

Use cases:

- Store properties of something in one place

Example:

```
my_particle = {  
    'x': 12.0,  
    'y': 13.5,  
    'z': -12.0,  
}
```

```
def print_particle(particle_dict):  
    print("x: ", particle_dict['x'])  
    print("y: ", particle_dict['y'])  
    print("z: ", particle_dict['z'])
```

# Data Structures

## Dictionaries

Use cases:

- Data you have to access by a name or ID
- Catalogs
- Filenames for datasets

Example:


```
30 runs_data_number = {  
31     'M29-768': 345  
32     'M35-1536': 138  
33     ...  
34 }  
35  
36 print(runs_data_number['M35-1536']) # 138  
37  
38 for run in dataset:  
39     run_number = runs_data_number[run]  
40     file = open(run + '/' + data_number, 'r')  
41     ...
```

# Data Structures

## Dictionaries – Trick!

You can use dictionaries to hold arguments for functions, apply them later

```
111 # Dictionaries: Tip for Function arguments
112 def myfunc(min, max, step):
113     pass
114
115 kwargs = {"min": 0, "max": 10, "step": 14}
116 myfunc(**kwargs)
117
118 all_the_args = [
119     {"min": 0, "max": 10, "step": 3},
120     {"min": 0, "max": 12, "step": 4},
121     {"min": 6, "max": 10, "step": 1}
122 ]
123 for kwargs in all_the_args:
124     myfunc(**kwargs)
125
```



# Data Structures

## Dictionaries – Tip!

You can use lists, and other dictionaries \*inside\* dictionary values!

```
28     # Dictionaries with lists for values
29     # Table of companions for some galaxy
30     galaxies = ["M31", "MW"]
31
32     companions = {}
33     companions["M31"] = ["M32", "M110", "etc"]
34     companions["MW"] = ["CMd", "LMC", "SMC", "etc"]
35
36     for galaxy in galaxies:
37         cs = companions[galaxy]
38         print(f"{galaxy} has {len(cs)} companions!")
39     # M31 has 3 companions!
40     # MW has 4 companions!
```



# Data Structures

## Dictionaries – Technique!

- If you're calling some function, e.g. from matplotlib lots of times with slight differences, use a dictionary to hold defaults

```
130 my_defaults = {  
131     "marker": "o",  
132     "color": "blue",  
133     "linewidth": 3,  
134     "alpha": 0.6  
135 }  
136  
137 plot([1, 2, 3], **my_defaults)  
138 plot([3, 4, 5], **my_defaults)
```

Better yet! Make a function!

# Data Structures

## Sets

- Sets are like a “**bag**” of values
- Values can be strings, numbers, tuples, etc.
- Values are unique.
  - Adding the same thing twice has no effect
- Not ordered
- “**instant**” adding and removing values
- “**instant**” membership testing
- Operations for
  - Testing
  - Grouping
  - Differencing
  - etc.

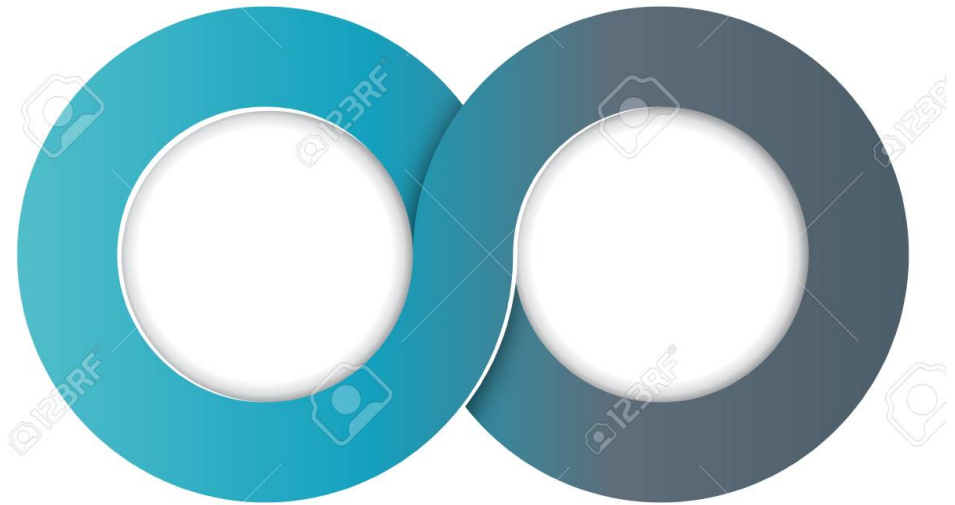
# Data Structures

## Sets

```
15 bar = set() # or {},}
16 bar.add(1)
17 bar.add(2)
18
19 print(2 in bar) # True
20 print(3 in bar) # False
21
22 baz = set([2,3,4])
23 print(bar & baz) # {2} in both
24 print(bar | baz) # {1,2,3,4} in either
25 print(bar ^ baz) # {1,3,4} in either, but not both!
26 print(bar > baz) # False is everythin in bar, also in baz?
```

Topic 2

# Iterators



## Topic 2

# Iterators

Generators create “lazy sequences”, called iterators

```
60     foo = range(0, 10000000000000, 1)
61     for b in foo:
62         print(b)
63         if b > 10:
64             break
```

This **works!**

Why doesn't my computer crash?

# Iterators

## Behind the scenes

Behind the scenes, they are a function that returns more than one value.

- Every time you run the function, you get the next value!

### Definition:

```
66 def count_down(stop):
67     value = 0
68     while value > stop:
69         yield value
70         value -= 1
```

### Usage:

```
72 for i in count_down(-5):
73     print(i, end=', ')
74 print()
75 # 0, -1, -2, -3, -4
```

# Iterators

## Useful Iterator Functions

- enumerate
  - Loop through things with indexes

```
78  # Iterators: enumerate
79  letters = ['a', 'b', 'c', 'd', 'e']
80
81  # Naive
82  for i in range(len(letters)):
83      letter = letters[i]
84      print(f"letter {letter} is at position {i}")
85
86  # Enumerate
87  for i, letter in enumerate(letters):
88      print(f"letter {letter} is at position {i}")
```

# Iterators

## Useful Iterator Functions

- zip
  - Group items from multiple lists into tuples

```
letters = ['a', 'b', 'c']
```

```
numbers = [5, 4, 3]
```

```
list(zip(letters, numbers))
```

```
# [('a', 5), ('b', 4), ('c', 3)]
```

```
for letter, number in zip(letters, numbers):
```

```
|     print(letter + str(number))
```

```
# a5
```

```
# b4
```

```
# c3
```



# Iterators

## Useful Iterator Functions

zip

- Trick! You can *unzip* as well, using \*

```
107     # Zip trick
108     print(zipped)           # [('a', 5), ('b', 4), ('c', 3)]
109     print(list(
110         |     zip(*zipped)   # (['a', 'b', 'c'], [1,2,3])
111         | ))
```

# Iterators

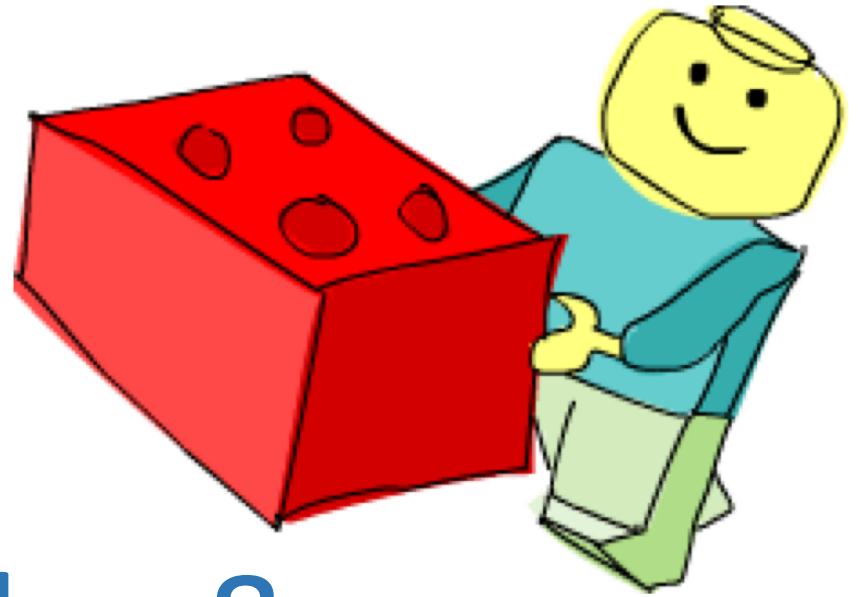
## Generator Expressions

- Generator comprehensions by example

```
8   # Make a sequence from 0 to 10^18
9   gen1 = range(1, 1000000000000000000)
10  # Square them all
11  gen2 = (i ** 2 for i in gen1)
12  # Include only odd numbers
13  gen3 = (i for i in gen2 if i % 2 == 0)
14  # Enumerate them all
15  gen4 = enumerate(gen3)
16
17  for i, value in gen4:
18      print(i, value)
19      if value > 120:
20          break
```

Prints:

0	4
1	16
2	36
3	64
4	100
5	144



Topic 3

# Functions, Files & Modules

## Topic 3

# Functions

## DRY: Do not Repeat Yourself

- Use functions to pull out repeated code, patterns

## Safety

- Variables inside functions can't **leak** out
- Easier variable naming

## Abstraction

- Once you've written one type of data analysis, don't need to worry about the details again

```
def fahr_to_celsius(temp):  
    return ((temp - 32) * (5/9))
```

The diagram illustrates the components of a Python function definition. Labels with arrows point to specific parts of the code:

- def statement**: Points to the `def` keyword.
- name**: Points to the function name `fahr_to_celsius`.
- parameter names**: Points to the parameter `temp` in the parentheses.
- body**: A bracket on the left side of the function body points to the `return` statement.
- return statement**: Points to the `return` keyword.
- return value**: Points to the expression `((temp - 32) * (5/9))`.

# Functions

## Rules of Thumb

### Create a function if:

- You write the same thing 2 or more times
- You know it will change in the future
- It was tricky to get right

```
theta = arctan2(sqrt(x**2 + y**2),z) * 180/np.pi  
phi = arctan2(y,x) * 180/np.pi + 180
```

- It's a step in a data analysis pipeline

```
data1    = read_in_data(...)  
data2    = read_in_data(...)  
merged   = match_data(data1, data2)  
reduced  = analyze_data(merged)  
plot_data(reduced)
```

# Functions

## Rules of Thumb (to avoid)

### “Magic Functions”

- It does different things, based on parameters

### Symptoms:

- 19+ parameters
- Some parameters can't be used with others
- matplotlib...

IMO scientists are particularly prone to this

Split it up!

# Functions

## Decorators

`@np.vectorize`

- Put this above your function, and it will work on arrays
- Lots of other use cases for decorators, but this is the one I use on a daily basis

# Functions: Docstrings

```
def galactocentric_distance(heliocentric_dist, galactic_long, galactic_lat):
```

```
    """\
```

```
    Calculate the galactocentric distance of an object
```

```
    ## INPUT
```

```
    * Heliocentric distance
```

```
    * galactocentric longitude
```

```
    * galactocentric latitude
```

```
    ## OUTPUT
```

```
    * Galactocentric distance (same as input units)
```

```
    Citation: Tobias Westmeier, 2018
```

```
    http://www.atnf.csiro.au/people/Tobias.Westmeier/tools\_coords.php
```

```
    ## EXAMPLES
```

```
    >>> galactocentric_distance(50.0, -57.2, -44.1)
```

```
    43.495214885473814
```

```
    >>> galactocentric_distance(13.0, -5.2, +19.1)
```

```
    11.464307213925464
```

```
    """
```

```
    component_1 = heliocentric_dist * math.cos(galactic_lat) * math.cos(galactic_long) - sun_galaxy_d
```

```
    component_2 = heliocentric_dist * math.cos(galactic_lat) * math.sin(galactic_long)
```

```
    component_3 = heliocentric_dist * math.sin(galactic_lat)
```

```
    galact_dist = (component_1 ** 2 + component_2 ** 2 + component_3)**0.5
```



## Topic 3

# Files & Modules

- Make a text file, give it a **.py** extension
- Congrats, you made a module!
- Any python file is a **module** that can be **imported**
- Python looks for modules in your current directory, and PYTHONPATH (that's where e.g. numpy is)
- If you want to find where a module is:
  - Import it, and **print(somemodule) .\_\_file\_\_**

## Topic 3

# Files & Modules

- Importing things should have no side-effects
- At the top level, modules should only include:
  - Constants
  - Functions
  - Classes
- Bad design: importing the module makes a plot show up

## Topic 3

# Packages

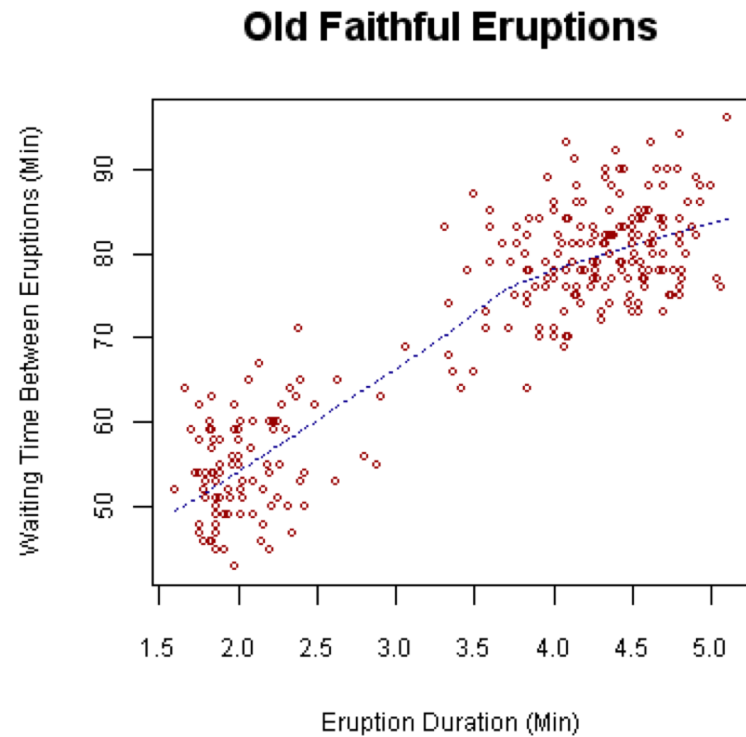
- Put a group of modules in a folder
  - Add a file named “\_\_init\_\_.py”
  - Congrats, you have a package!
- 
- Now you can group your modules together, e.g. scipy
  - You could even publish your package to pypi

# Exercises

1. Make a function that:
  - Takes three arguments (x,y,z)
  - Returns the distance of the vector
2. Make a function that:
  - Takes two points as dictionaries
  - Returns the distance between them
3. Put your functions in a module
4. Document them!
5. Import your module, and use it!

## Topic 4

# Plotting



# Available Packages

## matplotlib

- Standard

## Seaborn

- Great for statistics

## ProPlot

- Possibly good?

Check out the `matplotlib_basics.ipynb`  
and `plot_template.ipynb` files!

# Exercise

1. Open  
`star_formation_rate_exercise.ipynb`
2. Follow the instructions and use notebooks below to guide your plotting.

`matplotlib_basics.ipynb`  
`plot_template.ipynb`

3. We will go through step-by-step after some time.