A photograph of the Fermilab building, a large, modern, angular structure with a grid-like facade, set against a blue sky with scattered clouds. In the foreground, there are purple and yellow flowers.

An introduction to Python *for programmers*

Ruth Van de Water, Fermilab
Quantum Information Science Internship for Undergraduates
June 7, 2023

Why use Python?

Easy to learn

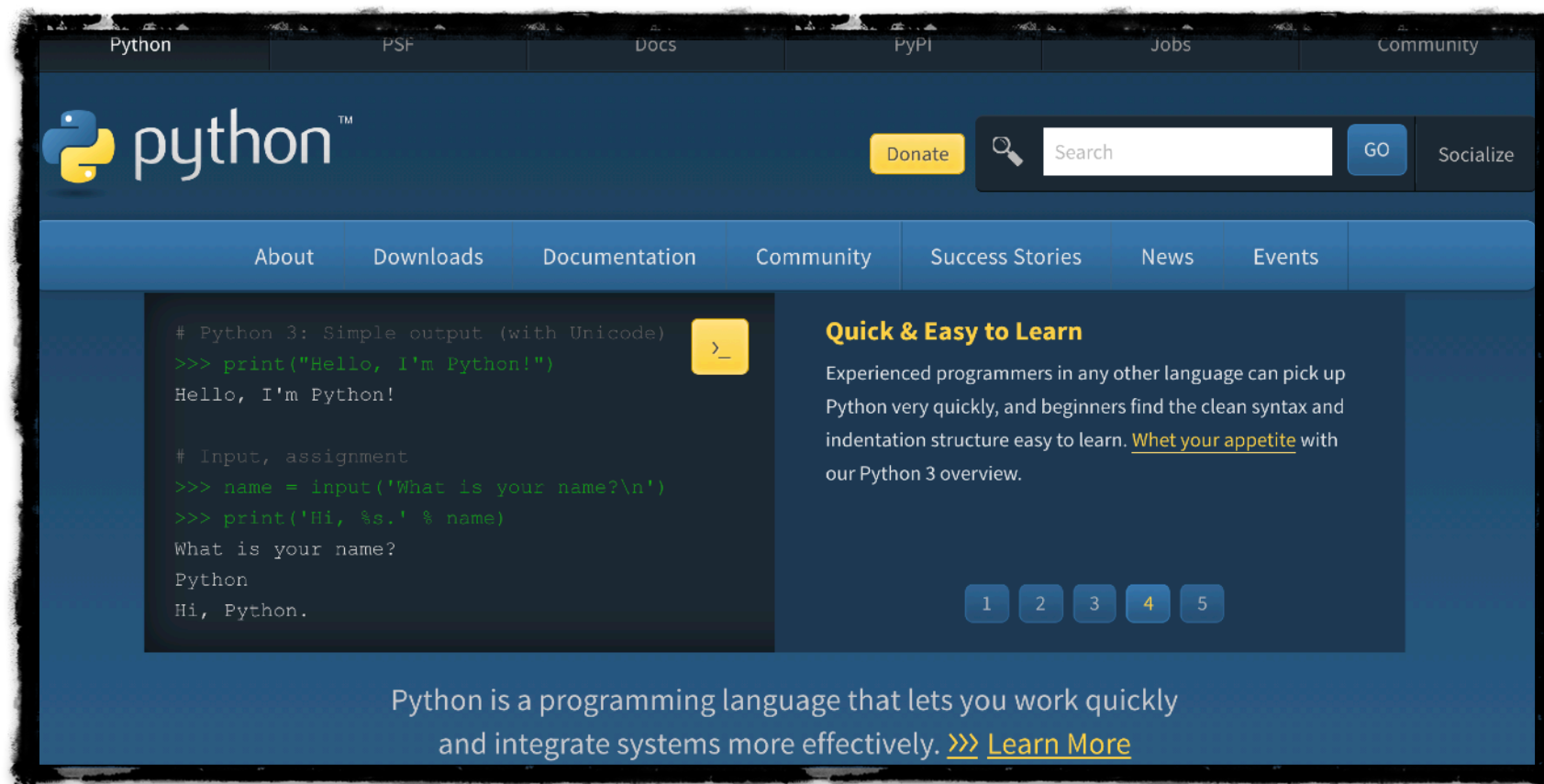
- Simple programming syntax, readable code, and English-like commands

Large developer community

- Well-maintained and documented
- Rich set of libraries for a wide variety of applications

Large user community

- Wealth of examples, tutorials, and other support available online



A bit about Python...

Python is a **scripting language**

- Code is *interpreted at runtime*, rather than compiled
- Code is **platform-independent** (will run on *any* machine that has same version of python installed)

Python is also a **high-level programming language**

- Has **robust standard library**
- *Object-oriented*: easy to create and use classes and objects
- Package support for **wide range of applications**:
 - Numerical analysis (*NumPy, Aesara*)
 - Scientific computing (*SciPy*)
 - Data processing (*Pandas*)
 - Plotting (*Matplotlib*)
 - Machine learning (*TensorFlow, Keras*)
 - Gaming (*PyGame*)



Getting started with Python

Official Python website (<https://www.python.org/>)

Python3 documentation (<https://docs.python.org/3/>)

w3 Schools (<https://www.w3schools.com/python/>)

- Tutorials and examples for all of the basics*

Stack Overflow (<https://stackoverflow.com/>)

- *Community-sourced collection of coding questions & answers*

... but the fastest way to find the answer to your question is usually to just  oogle it!

*Most of the examples in this talk are from w3 Schools' Python lessons

Outline

- The basics

- print() function
- Dynamical typing
- ints & floats

- Lists and other data types

- Indexing & slicing
- Methods (append, delete, insert, reverse, ...)
- Iterating (for loops)
- enumerate() & range() functions
- Arrays, dictionaries, sets, ...

- Conditions

- Comparison operators
- if, else, elif
- while loops
- break & continue

- Functions

- Creating & using
- Local & global variables

- Exception Handling

The basics

Hello world!

We are using Python 3.
Python 2.7 is no longer supported.

The `print()` function

The `print()` function prints the specified message to the screen, or other standard output device.

The message can be a string, or any other object, the object will be converted into a string before written to the screen.

```
Jupyter QtConsole
Jupyter QtConsole 4.7.5
Python 3.8.3 (default, Jul 2 2020, 11:26:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print( 'Hello, world!' )
Hello, world!

In [2]: name = 'Ruth'

In [3]: age = 42

In [4]: print( 'My name is {} and I am {} years old.'.format(name,age) )
My name is Ruth and I am 42 years old.
```

Can use single or double quotes around a string

Can print *any* Python object: strings, ints, floats, lists, dictionaries, ...

Numeric types

Python supports **integers**, **floating-point**, and **complex** numbers

- Uses **dynamical typing**, *i.e.*, variable types are determined only at runtime

This code stores 3.14159 in memory and binds the name pi to it. After it runs, the type of pi is a **float**.

```
In [12]: pi = 3.14159
```

```
In [13]: type(pi)
```

```
Out[13]: float
```

```
In [14]: int(pi)
```

```
Out[14]: 3
```

The **int()** function removes all digits after the decimal.

```
In [15]: pi = 3.14159 + 0j
```

```
In [16]: type(pi)
```

```
Out[16]: complex
```

A **j** to the right of a number makes it **imaginary**.

Jupyter QtConsole

```
In [34]: print(G3)
6.674e-11
```

To **exponentiate** a number, use the **** operator** or the **pow()** function

```
In [35]: G = 6.674 * 10**(-11)
```

```
In [36]: G2 = 6.674 * pow(10, -11)
```

```
In [37]: G3 = 6.674e-11
```

Python also knows **scientific notation**

```
In [38]: print( 'G = {}, G2 = {}, G3 = {}'.format(G,G2,G3) )
G = 6.674e-11, G2 = 6.674e-11, G3 = 6.674e-11
```

Lists and other data types

Lists, indexing, and slicing

Lists are used to store multiple items in a single variable.

Lists are created using square brackets.

List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

```
Jupyter QtConsole
In [80]: mylist = ['item 1', 'item 2', 'item 3', 'item 4']
In [81]: type(mylist)
Out[81]: list
In [82]: len(mylist)
Out[82]: 4
In [83]: mylist[0]
Out[83]: 'item 1'
In [84]: mylist[3]
Out[84]: 'item 4'
In [85]: mylist[-1]
Out[85]: 'item 4'
In [86]: mylist[-2]
Out[86]: 'item 3'
```

The `len()` function gives the number of list entries.

List items are also indexed from the end. The *last* item has index `[-1]`.

```
In [98]: mylist[1:3]
Out[98]: ['item 2', 'item 3']
In [98]:
In [99]: mylist[:3]
Out[99]: ['item 1', 'item 2', 'item 3']
In [99]:
In [100]: mylist[1:]
Out[100]: ['item 2', 'item 3', 'item 4']
In [101]:
```

Can also specify a *range of items* between two indices.

The *last index is not included!*

Omit the first index to start at zero.

Omit the second index to go to the end of the list.

List methods

- Python objects can possess **methods**. Methods are *functions that belong to that object*.

Methods that can be used on Python lists

Method	Description
<code>append()</code>	Adds an element at the end of the list
<code>clear()</code>	Removes all the elements from the list
<code>copy()</code>	Returns a copy of the list
<code>count()</code>	Returns the number of elements with the specified value
<code>extend()</code>	Add the elements of a list (or any iterable), to the end of the current list
<code>index()</code>	Returns the index of the first element with the specified value
<code>insert()</code>	Adds an element at the specified position
<code>pop()</code>	Removes the element at the specified position
<code>remove()</code>	Removes the item with the specified value
<code>reverse()</code>	Reverses the order of the list
<code>sort()</code>	Sorts the list

List methods

- Methods *may change the state of an object*.

```
Jupyter QtConsole
In [108]: fruit = ['apple', 'banana', 'cherry']
In [109]: fruit.append('orange')
In [110]: print(fruit)
['apple', 'banana', 'cherry', 'orange']
In [111]: fruit.insert(3, 'grape')
In [112]: print(fruit)
['apple', 'banana', 'cherry', 'grape', 'orange']
In [113]: fruit.reverse()
In [114]: print(fruit)
['orange', 'grape', 'cherry', 'banana', 'apple']
In [115]: |
```

The **append()** method adds an item to the end of the list.

The **insert()** method adds an item at the specified location.

The **reverse()** method reverses the order of a list.

Looping over lists

Can use **for loops** to fill lists or to run over list entries.

- Python uses **indentation** (rather than `{}` or `begin/end`) *to show nesting*.

```
Jupyter QtConsole
In [130]: fruit = ['apple', 'banana', 'cherry', 'grape', 'orange']

In [131]: for f in fruit:
...:     print(f)

apple
banana
cherry
grape
orange
```

The *first line* of a **for** loop must end with a colon.
The body of a **for** loop *must* be indented.

```
Jupyter QtConsole
In [123]: numbers = []

In [124]: for i in range(1,11):
...:     numbers.append(i)

In [125]: print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The **range(start, stop)** function creates a series from *start* to *stop-1*.

```
Jupyter QtConsole
In [135]: for count, f in enumerate(fruit):
...:     print(count, f)

0 apple
1 banana
2 cherry
3 grape
4 orange
```

The **enumerate()** function is useful when you need the value *and the count* in a for loop.

Other useful data types

Python has **four collection data types**

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered* and changeable. No duplicate members.

```
Jupyter QtConsole
In [145]: mydict = {
...:     "first": "Ruth",
...:     "last": "Van de Water",
...:     "month": "October",
...:     "day": 8,
...:     "year": 1978
...: }

In [146]: for key in mydict:
...:     print( '{}: {}'.format(key,mydict[key]) )
first: Ruth
last: Van de Water
month: October
day: 8
year: 1978
```

Dictionaries store data in *key:value pairs*. Keys can be almost *any type*: integers, floats, strings, tuples, ...

You can also iterate over a dictionary.

The flexibility of dictionary keys makes dictionaries very useful!

Conditions

Comparisons

When two values are compared, Python evaluates the expression and returns the Boolean answer, *i.e.*, **True** or **False**

```
In [148]: print(10 > 9)
True

In [149]: print(10 == 9)
False

In [150]: print(10 >= 9)
True

In [151]: print(bool('abc'))
True

In [152]: print(bool([1,2,3]))
True

In [153]: print(bool([]))
False
```

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Almost *any* Python object evaluates to **True** if it is not zero or empty

if, else, and elif

Logical conditions are most often used in **if** statements and **while** loops

```
Jupyter QtConsole
In [161]: favorite = "Sylveon"
In [162]: if (favorite == "Pikachu"):
...:     print("Pika, pika!")
...: elif (favorite == "Eevee"):
...:     print("Eeeeeveeeee!")
...: else:
...:     print("There are other Pokemon?")
There are other Pokemon?
```

Each if/else/elif statement must *end with a colon*.
The code to be executed if the condition is met must be *indented*.

```
Jupyter QtConsole
In [169]: t=0
In [170]: while (t<=5):
...:     print(t)
...:     t = t+1
0
1
2
3
4
5
```

The *first line* of a **while** loop must *end with a colon*.
The body must be *indented*.

break and continue

To *break out* of a **for** or **while** loop, use the **break** statement.

To *skip an iteration* of a **for** or **while** loop, use the **continue** statement.

```
Jupyter QtConsole  
In [180]: instructors=['Hank', 'Mike', 'Ruth']  
  
In [181]: for i in instructors:  
...:     if (i == 'Mike'):  
...:         break  
...:     else:  
...:         print(i)  
Hank  
  
In [182]: for i in instructors:  
...:     if (i == 'Mike'):  
...:         continue  
...:     else:  
...:         print(i)  
Hank  
Ruth
```

When Python sees **break**, it exits the loop and moves on to the next line in the code

When python sees **continue**, it skips this iteration of the loop and moves on to the next list element.

Modules and namespaces

Modules

Python users can save *definitions* of constants, functions, and other related objects in separate **Module** files

- File name is module name plus `.py` suffix
- Module definitions must be *imported* before use
- Can easily download and install most modules not included in default Anaconda using *package managers* **pip** or **Conda**

Can introduce naming conflicts and is not recommended!

Import *all* definitions in the numpy module. Definitions are specified with the `numpy.prefix`.

Import all definitions in the numpy module with a `*`. Prefixes are not required.

Import *only* definitions that are needed. Prefixes are not required.

```
In [5]: import numpy
In [6]: pi = numpy.pi
In [7]: numpy.cos(pi/4)
Out[7]: 0.7071067811865476
```

```
In [5]: from numpy import *
In [6]: cos(pi/4)
Out[6]: 0.7071067811865476
In [7]:
```

```
In [8]: from numpy import pi, cos
In [9]: cos(pi/4)
Out[9]: 0.7071067811865476
In [10]:
```

Namespaces

A **namespace** is a mapping from names to object

- Python uses namespaces to organize the symbolic names given to variables, functions, and other objects within a program, and to avoid naming conflicts.
- Python functions Examples of namespaces are: the set of built-in names (containing functions such as [`print()` and `range()`]); the global names in a module; and the local names in a function invocation. ... [T]here is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion [because] users of the modules must prefix it with the module name. *cannot change the state of an object*
- *In Python, `_` "a namespace is a mapping from names to objects." `_` ItSome namespaces that you have already encountered are*
- *- the `**`*

Functions

Functions

Functions are blocks of code that are *only* executed when they are *called*

- You can pass objects (=parameters) to a Python function
- Python functions can return objects (=output) as a result
- Python functions *cannot change the state of an object*

Functions begin with the **def** keyword

The **arguments** of the function are given in parentheses immediately following the name. *Functions are not required to have arguments.*

```
In [192]: def countlist(mylist):  
          count = 0  
          for m in mylist:  
              count = count+1  
          return count
```

The *body* of the function is *indented*

The *first line* of a function ends with a *colon*.

Use the **return** statement to return objects from the function.

```
In [193]: numbers = list(range(1,11))
```

```
In [194]: print(numbers)  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [195]: countlist(numbers)  
Out[195]: 10
```

Scope

A Python variable can only be used within the **scope** in which it was *defined*

- A variable created *within a function* is **local** to that function and can only be accessed within that function
- A variable created *in the main body* of the code is **global** and can be accessed anywhere in the code (*after it has been created*)
- If the same variable name is used both inside and outside a function, Python treats them as two separate variables — one which is available in the local scope of the function and the other which is available in the global scope of the program
- If a variable is provided to the function as an argument or defined within the function, Python will look for the variable out

Exception handling

Exceptions

- Sometimes — even when all syntax is correct — a line of *Python code may generate an error at runtime*

```
In [197]: def y(x):
...:     return 1/x

In [198]: y(100)
Out[198]: 0.01

In [199]: y(0)

-----

ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-199-04f3f9254374> in <module>
----> 1 y(0)

<ipython-input-197-1817af668cb5> in y(x)
      1 def y(x):
----> 2     return 1/x

ZeroDivisionError: division by zero
```

Although $y=1/x$ is a perfectly good function, it *fails with a ZeroDivision Error when $x=0$*

```
Jupyter QtConsole

In [206]: from math import cos, pi

In [207]: def y(x):
...:     return cos(x)

In [208]: y(pi)
Out[208]: -1.0

In [209]: y('pi')

-----

TypeError                                Traceback (most recent call last)
<ipython-input-209-6f4a466a7707> in <module>
----> 1 y('pi')

<ipython-input-207-b0220e16ec7b> in y(x)
      1 def y(x):
----> 2     return cos(x)

TypeError: must be real number, not str
```

There are no problems $y=\cos(x)$, but it *fails with a TypeError when x is a string*

Errors that arise during the execution of Python code are called **exceptions**

Common Python exceptions

Exceptions can arise in otherwise working Python code when certain conditions are met. For example:

exception **FileNotFoundError**

Raised when a file or directory is requested but doesn't exist. Corresponds to errno `ENOENT`.

exception **IndexError**

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, `TypeError` is raised.)

exception **KeyError**

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception **TypeError**

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception **ValueError**

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

exception **ZeroDivisionError**

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

try and except

- Without intervention, Python terminates a program immediately when an exception occurs.
- However, Python allows the programmer to *catch anticipated exceptions and handle them* so that the program continues to operate smoothly when such special cases arise. This is done with **try and except**.

```
Jupyter QtConsole

In [240]: KantoStarters = {
...:     'fire': 'Charmander',
...:     'water': 'Squirtle',
...:     'grass': 'Bulbasaur'}

In [241]: type = 'grass'

In [242]: try:
...:     pokemon = KantoStarters[type]
...: except KeyError:
...:     print('Try again. Possible Kanto starter is ' +
'water')
...: else:
...:     print('The {}-type starter Pokemon in the Kanto region is
{}'.format(type, pokemon))

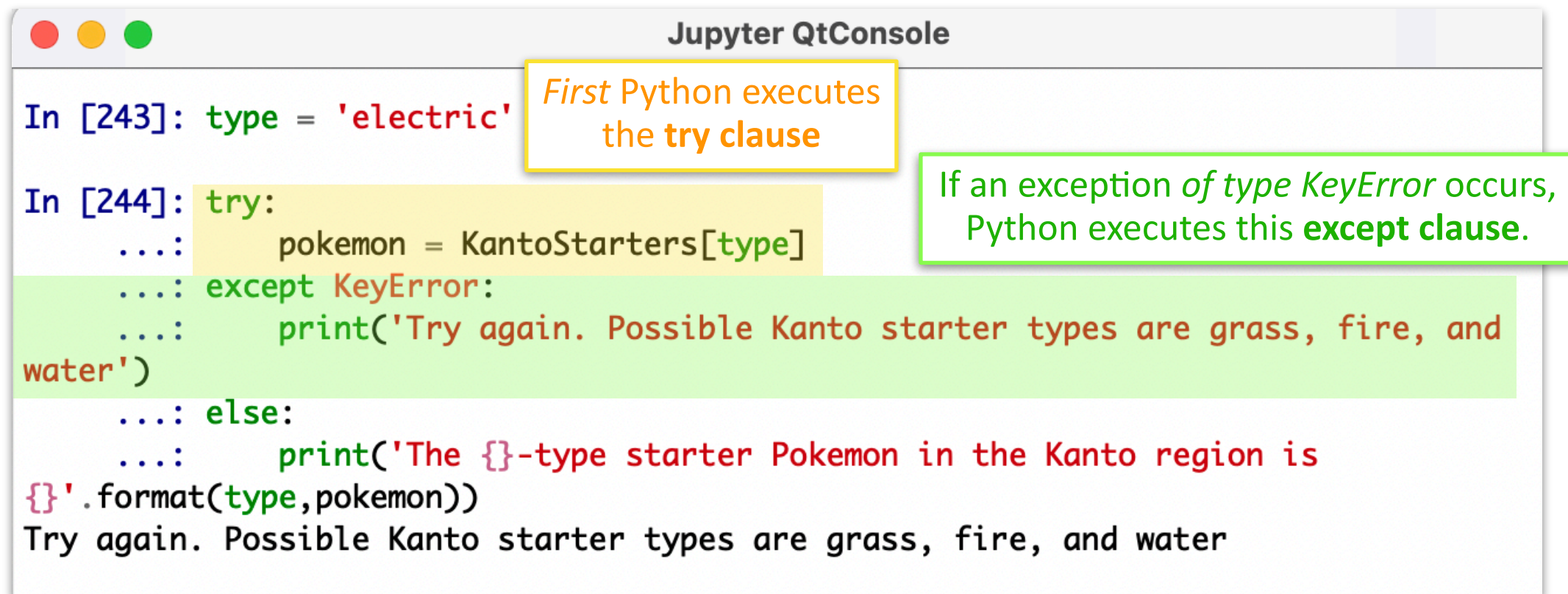
The grass-type starter Pokemon in the Kanto region is Bulbasaur
```

First Python executes the try clause

If no KeyError exception occurs, Python skips the except clause and executes the else clause

try and except

- Without intervention, Python terminates a program immediately when an exception occurs.
- However, Python allows the programmer to *catch anticipated exceptions and handle them* so that the program continues to operate smoothly when such special cases arise. This is done with **try and except**.



```
In [243]: type = 'electric'
In [244]: try:
...:     pokemon = KantoStarters[type]
...: except KeyError:
...:     print('Try again. Possible Kanto starter types are grass, fire, and
water')
...: else:
...:     print('The {}-type starter Pokemon in the Kanto region is
{}'.format(type, pokemon))
Try again. Possible Kanto starter types are grass, fire, and water
```

First Python executes the **try clause**

If an exception of type *KeyError* occurs, Python executes this **except clause**.

- Although try and except are not band-aids for patching up poorly written code, they are *versatile and handy tools in a Python programmer's toolkit*

Good coding practices

Take responsibility

- At the top of your code, as comments, list the author's name and contact information, the date when the code was written, and a one or two sentence summary of what the code does. For example:

```
# Hello.py
# This program writes a friendly greeting to the screen.
#
# Author: Ruth Van de Water (rsvandewater@noctrl.edu)
# Last revised: March 22, 2019
```


Be as clear as possible

Make sure that any output from your program is **self-explanatory**

- Use *labels on graphs* and *words in print statements* to identify what is being shown. For example:

Do print: “Final position of projectile is x=15 m, y=0 m”

Don't simply print: “15, 0”

Use **consistent** and *appropriate* variable names

- If you are making a ball name it **Ball** or **Projectile** or **Spacecraft** (*depending upon the physical problem*).
- If you are making several balls, name them **Ball_1**, **Ball_2**, and **Ball_3** or **Red_Ball**, **Green_Ball**, and **Yellow_Ball**.
- Naming the the objects in your code **Doritos**, **Cheetos**, and **Fritos**¹ may *seem* funny at the time, but will only lead to confusion when you're writing and debugging your code.

¹Yes, a student actually *did* this in an assignment.

Avoid duplication

- If you find that you need to copy-and-paste a particular grouping of code more than once or twice, write a function for the task that you can call when needed. Even though it sometimes seem like a waste of time to write a function for a simple job, it's actually more efficient and less error-prone.
- Define each constant *only once at the top of your code*; then use the variable name in the rest of the program. For example:

```
g = 9.80 #m/s^2  
m_E = 5.9722 * 10**{24} #Mass of Earth in kg  
hbar = 6.62607015 * 10**{-34} #J*s
```
- This way — if you need to change a numerical value or modify a function — you only have to change your code in *one place*, rather than several.

Comment, comment, *comment!*

- Not only may our code be used by another individual at a later date, but often in research we need revisit our code weeks or months later and find that we have forgotten the details. Therefore, our *code's logic must be clear, with its key elements clearly identified and explained.*
- Use comments to:
 - Identify the main steps in your code
 - Describe the **Think first, then code!** how to use them,
 - Specify the units of all quantities that have them,
 - Add references for numbers, equations, or algorithms that are not common knowledge, ...
- **Always comment “clever” solutions!** Otherwise you will be puzzled months later trying to understand what it is that you’re doing, and why you chose that approach or implementation as opposed to something simpler



Q: What is the *most important* Python function for debugging coding errors?

A: The **print()** function. You can use print to check if a value is correct or whether you passed an if statement or whether you're stuck in a loop or just about anything ...



Q: What should you do if you've forgotten how to do something in Python?



Q: Where should you look if you want to know how to do something new in Python?



Q: Should you panic if you get an inscrutable* error message from Python?

A: NO!



*impossible to understand or interpret.

Questions?

