SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Spring 2024

If we have data, let's look at data. If all we have are opinions, let's go with mine.

— Jim Barksdale

Introduction

Please introduce yourself in 2 minutes.

1. Your name
2. Your undergrad and grad majors
3. Your background in python programming, data analytics/science, and artificial intelligence
4. What you hope to get out of this class
5. Any fun thing you want us to know

What Kinds of Data?

- **Defining Structured Data**: A broad term encompassing various common data forms.
- **Tabular Data**: Data in tables where each column represents a different type (string, numeric, date, etc.), commonly used in relational databases and text files.
- **Multidimensional Arrays**: Data organized in matrices, essential for complex numerical computations.
- **Relational Data**: Multiple tables connected by key columns, akin to primary or foreign keys in SQL.
- **Time Series Data**: Data points collected over time, which can be evenly or unevenly spaced.
- **Transforming and Extracting Data**: Many datasets can be converted into structured forms for analysis, such as turning news articles into word frequency tables for sentiment analysis.

Why Python for Data Analysis?

- **Python's Appeal**: Popular since 1991, Python is one of the most used interpreted languages.
- **Evolution in Web Development**: Gained popularity with frameworks like Django for web development since 2005.
- **Beyond Scripting**: Python's role extends beyond scripting to serious software development.
- **Scientific Computing and Data Analysis**: Python has a large, active community in these fields.
- **Data Science and Machine Learning**: Python has evolved into a key language in these domains.

Why Python for Data Analysis? Cont'd

- **Comparison with Other Languages**: Python is often compared with R, MATLAB, SAS, and Stata.
- **Python as Glue**: Effective in integrating C, C++, and FORTRAN code in scientific computing.
- **Two-Language Problem**: Python's role in both research and production phases in organizations.
- **Emergence of JIT Compiler Technology**: Libraries like Numba enhance Python's performance in computational algorithms.
- **Python's Organizational Benefits**: Shared tools among researchers and software engineers enhance efficiency.

Why Not Python?

- **Performance**: Python, being an interpreted language, generally runs slower than compiled languages like Java or C++.
- **Trade-off**: Python's ease of use often outweighs its slower execution time, except in low-latency or resource-intensive applications.
- **Concurrency Challenges**: Python struggles with highly concurrent, multithreaded applications due to the Global Interpreter Lock (GIL).
- **GIL Limitation**: The GIL prevents executing more than one Python instruction at a time, limiting Python's efficiency in certain multithreaded scenarios.
- **Big Data Processing**: For large-scale data processing, clusters of computers are often required to compensate for Python's performance limitations.
- **Native Extensions**: Python can achieve true multithreading through C extensions that bypass the GIL, suitable for parallel code execution.

Essential Python Libraries: NumPy

- **Core of Numerical Computing**: NumPy is integral to Python's scientific computing capabilities.
- **Multidimensional Arrays**: Features the efficient ndarray for handling large, multi-dimensional datasets.
- **Element-wise Computations**: Supports operations both within and between arrays for complex calculations.
- **Data I/O Tools**: Provides tools for handling array-based data storage and retrieval.
- **Advanced Mathematical Functions**: Includes functions for linear algebra, Fourier transforms, and random number generation.
- **Mature C API**: Facilitates integration with Python extensions and native C/C++ code for accessing NumPy structures.
- **Efficiency and Interoperability**: More efficient than built-in Python structures for numerical data, and allows operation by lower-level languages without data copying.

Essential Python Libraries: Pandas

- **High-Level Data Structures**: Pandas offers intuitive structures for structured or tabular data.
- **Emergence and Impact**: Since 2010, pandas has significantly enhanced Python's capabilities in data analysis.
- **Primary Objects**: DataFrame for tabular data and Series for one-dimensional labeled arrays.
- **Blending NumPy and Data Manipulation**: Combines NumPy's array computing with spreadsheet-like data manipulation.
- **Versatile Data Handling**: Features like reshaping, slicing, aggregation, and subsetting data.
- **Origins**: Developed at AQR Capital Management to address specific data analysis needs in finance.

Essential Python Libraries: Pandas

- **Labeled Axes and Data Alignment**: Facilitates error-free handling of misaligned or differently indexed data.
- **Time Series Functionality**: Equally adept at handling time series and non-time series data.
- **Community Development**: Evolved into a community-maintained project with a wide contributor base.
- **Comparison with R**: DataFrame concept in pandas is akin to data frames in R, making it familiar to R users.
- **Name Origin**: 'pandas' is a play on 'panel data' and 'Python data analysis'.

Essential Python Libraries: Matplotlib

- **Popular Visualization Library**: Matplotlib is the leading Python library for 2D data visualizations.
- **Origin and Maintenance**: Developed by John D. Hunter and maintained by a large team.
- **Publication-Quality Plots**: Designed to create high-quality, publishable plots.
- **Integration with Python Ecosystem**: Works well with other Python libraries, making it a versatile tool in data analysis.
- **Widespread Usage**: Remains a widely-used library despite the emergence of newer visualization tools.
- **Safe Default Choice**: Considered a reliable and effective option for general visualization needs.

Essential Python Libraries: IPython and Jupyter

- **IPython's Inception**: Began in 2001 as Fernando Pérez's project for an improved Python interpreter.
- **Evolution into Essential Tool**: Grew into a key tool for interactive computing and software development in Python.
- **Interactive and Exploratory Workflow**: Facilitates an execute-explore approach, ideal for data analysis.
- **Jupyter Project Expansion**: In 2014, IPython evolved into the Jupyter project, supporting over 40 programming languages.
- **IPython as Jupyter Kernel**: Functions as a Python execution kernel within the Jupyter notebook system.
- **Rich Content Creation**: Jupyter notebooks enable creation of documents combining code, Markdown, and HTML.

Essential Python Libraries: SciPy

- **SciPy Overview**: A comprehensive library for foundational scientific computing.
- **Integration and Solvers**: Module 'scipy.integrate' offers numerical integration and differential equation solvers.
- **Advanced Linear Algebra**: 'scipy.linalg' extends beyond 'numpy.linalg' with more sophisticated linear algebra routines and decompositions.
- **Optimization Tools**: 'scipy.optimize' provides function optimizers and algorithms for root finding.
- **Signal Processing**: The 'scipy.signal' module contains tools for signal processing.
- **Statistics and Probability**: 'scipy.stats' includes probability distributions, statistical tests, and descriptive statistics tools.
- **Comprehensive Scientific Toolkit**: SciPy, in conjunction with NumPy, forms a robust foundation for traditional scientific computing tasks.

Essential Python Libraries: scikit-learn

- **scikit-learn's Role in Machine Learning**: A leading toolkit for machine learning in Python, established in 2007.
- **Community Contributions**: Developed with the input of over two thousand individual contributors.
- **Broad Range of Models**: Includes models for classification (e.g., SVM, random forest), regression (e.g., Lasso), and clustering (e.g., k-means).
- **Advanced Machine Learning Techniques**: Features tools for dimensionality reduction (e.g., PCA), model selection (e.g., grid search), and preprocessing (e.g., normalization).
- **Integration with Python Data Ecosystem**: Works seamlessly with pandas, statsmodels, and IPython, enhancing Python's capabilities in data science.

Essential Python Libraries: statsmodel

- **Development of statsmodels**: Originated from Jonathan Taylor's work at Stanford, formalized in 2010 by Skipper Seabold and Josef Perktold.
- **Patsy Project Integration**: Incorporates Nathaniel Smith's Patsy for model specifications, inspired by R's formula system.
- **Focus on Classical Statistics**: Specializes in frequentist statistics and econometrics, contrasting with scikit-learn's prediction focus.
- **Comprehensive Statistical Analysis**: Includes regression models, ANOVA, time series analysis, and nonparametric methods.
- **Statistical Inference and Visualization**: Emphasizes statistical inference, uncertainty estimates, and visualizing model results.
- **Compatibility with Python Ecosystem**: Designed to work seamlessly with NumPy and pandas for data analysis.

Essential Python Libraries: others

- **TensorFlow and PyTorch**: Leading libraries for deep learning and machine learning, essential for AI development.
- **LangChain**: A newer tool specializing in language models and natural language processing, facilitating advanced AI applications.
- **General Data Wrangling**: Emphasis on foundational skills in data manipulation and analysis with Python.
- **Foundation for Advanced Learning**: Recommended as a starting point before progressing to more specialized AI and machine learning resources.
- **Diverse Python Ecosystem**: Python's extensive library support caters to a wide range of data science and AI needs.
- **Readiness for Specialization**: Preparation through general-purpose tools to engage with more complex and specialized AI libraries.

Essential Python Libraries: others

- **OpenCV**: A robust library for computer vision and image processing, widely used in real-time applications.
- **Streamlit**: Enables the creation of interactive and shareable web applications for machine learning and data science.
- **Gradio**: A library for building customizable UI components for machine learning models, enhancing model testing and sharing.
- **PyCaret**: An automated machine learning library that simplifies the workflow of deploying machine learning models.
- **Seaborn**: A visualization library based on matplotlib, offering high-level interfaces for drawing attractive statistical graphics.
- **Bokeh**: Specialized in interactive and real-time streaming visualizations, suitable for modern web browsers.
- **Keras**: A high-level neural networks API, known for its user-friendliness, modularity, and extensibility, running on top of TensorFlow.

Integrated Development Environments and Text Editors

- Preferred development environment often combines IPython with a text editor for iterative testing and debugging in Python.
- Interactive data manipulation and verification is facilitated using tools like IPython or Jupyter notebooks, along with libraries like pandas and NumPy.
- Richly featured IDEs are preferred by some for more comprehensive development needs, offering more than minimal text editors like Emacs or Vim.
- Examples of popular IDEs include PyDev (free, Eclipse-based), PyCharm (JetBrains, subscription-based), Python Tools for Visual Studio, Spyder (free, shipped with Anaconda), and Komodo IDE (commercial).
- Modern text editors like VS Code and Sublime Text 2 are also widely used due to their excellent support for Python programming.

Executing Python Statements

You can execute arbitrary Python statements by typing them and pressing Return (or Enter). When you type just a variable into Jupyter, it renders a string representation of the object:

```
[2]: import numpy as np
     data = [np.random.standard_normal() for i in range(7)]
     data
```

```
[2]: [0.4967141530112327,
      -0.13826430117118466,
      0.6476885381006925,
      1.5230298564080254,
      -0.23415337472333597,
      -0.23413695694918055,
      1.5792128155073915]
```

Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

```
[3]: a = [1, 2, 3]

[4]: b = a
     b

[4]: [1, 2, 3]

[5]: b?

Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

[6]: print?

Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

```
[7]: def add_numbers(a, b):
         """
         Add two numbers together

         Returns
         -------
         the_sum : type of arguments
         """
         return a + b
```

```
[8]: add_numbers?
```

```
Signature: add_numbers(a, b)
Docstring:
Add two numbers together

Returns
-------
the_sum : type of arguments
File:      /var/folders/g9/05llvmcn0r5ghq_b99qscb_w0000gn/T/ipykernel_34798/1411870314.py
Type:      function
```

Variables and argument passing

When assigning a variable (or name) in Python, you are creating a reference to the object shown on the righthand side of the equals sign. In practical terms, consider a list of integers:

```
[9]: a = [1, 2, 3]
```

```
[10]: b = a
      b
```

```
[10]: [1, 2, 3]
```

```
[11]: a.append(4)
      b
```

```
[11]: [1, 2, 3, 4]
```

Variables and argument passing

- Passing Objects to Functions: Objects passed as arguments create new local variables that reference the original objects without copying.
- Scope and Variable Binding: Binding a new object to a variable inside a function doesn't affect a variable with the same name in the parent scope.
- Mutability of Arguments: Possible to alter the internals of a mutable argument passed to a function.
- Function Behavior Example: Understanding the impact of this behavior requires considering specific function examples and how they manipulate their arguments.

```
[12]: def append_element(some_list, element):
          some_list.append(element)
```

```
[13]: data = [1, 2, 3]
      append_element(data, 4)
      data
```

```
[13]: [1, 2, 3, 4]
```

Dynamic references, strong types

Variables in Python have no inherent type associated with them; a variable can refer to a different type of object simply by doing an assignment

```
[14]: a = 5
      type(a)
      a = "foo"
      type(a)

[14]: str

[15]: "5" + 5

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 "5" + 5

TypeError: can only concatenate str (not "int") to str
```

Dynamic references, strong types

- No Implicit Casting: Python does not allow implicit conversion between data types like converting '5' to 5 or vice versa.
- Strongly Typed Language: Python is strongly typed, meaning every object has a specific type, and implicit type conversions are limited.
- Permitted Conversions: Implicit conversions in Python occur only in specific, permissible situations.

```
[16]: a = 4.5
      b = 2
      # String formatting, to be visited later
      print(f"a is {type(a)}, b is {type(b)}")
      a / b

      a is <class 'float'>, b is <class 'int'>

[16]: 2.25
```

Here, even though b is an integer, it is implicitly converted to a float for the division operation.

Type Knowledge

- Importance of Type Knowledge: Essential to know an object's type in Python, especially for writing versatile functions.
- Using `isinstance` Function: Allows checking if an object is an instance of a specific type.
- Tuple for Multiple Types: `isinstance` can accept a tuple of types to check if an object belongs to any of the types in the tuple.

```
[17]: a = 5
      isinstance(a, int)

[17]: True

[18]: a = 5; b = 4.5
      isinstance(a, (int, float))
      isinstance(b, (int, float))

[18]: True
```

Attributes and Methods

- Objects with Attributes and Methods: In Python, objects usually possess both attributes (objects stored within) and methods (functions associated with the object).
- Accessing Attributes and Methods: Use the syntax `<obj.attribute_name>` to access an object's attributes or methods.
- Interaction with Internal Data: Methods can interact with an object's internal data, providing functionality specific to the object.

```
[19]: a = "foo"

[ ]: a.<tab>

[20]: getattr(a, "split")

[20]: <function str.split(sep=None, maxsplit=-1)>
```

Duck Typing

- Concept of Duck Typing: In Python, focusing on an object's behavior or methods rather than its type, as per the saying "If it walks like a duck and quacks like a duck, then it's a duck."
- Checking for Methods and Behaviors: More importance is given to whether an object implements certain methods or behaviors than to its actual type.
- Verifying Object Iterability: An object is considered iterable if it implements the iterator protocol, typically through the `__iter__` method or by successfully passing the `iter` function test.

Duck Typing

```
[21]: def isiterable(obj):
          try:
              iter(obj)
              return True
          except TypeError: # not iterable
              return False
```

```
[23]: isiterable("a string")
```

```
[23]: True
```

```
[24]: isiterable([1, 2, 3])
```

```
[24]: True
```

```
[25]: isiterable(5)
```

```
[25]: False
```

Binary operators

Most of the binary math operations and comparisons use familiar mathematical syntax used in other programming languages:

```
[27]: 5 - 7

[27]: -2

[28]: 12 + 21.5

[28]: 33.5

[29]: 5 <= 2

[29]: False
```

Binary operators

Operation	Description
`a + b`	Add `a` and `b`
`a - b`	Subtract `b` from `a`
`a * b`	Multiply `a` by `b`
`a / b`	Divide `a` by `b`
`a // b`	Floor-divide `a` by `b`, dropping any fractional remainder
`a ** b`	Raise `a` to the `b` power
`a & b`	`True` if both `a` and `b` are `True`; for integers, take the bitwise `AND`
`a | b`	`True` if either `a` or `b` is `True`; for integers, take the bitwise `OR`
`a ^ b`	For Booleans, `True` if `a` or `b` is `True`, but not both; for integers, take the bitwise `EXCLUSIVE-OR`
`a == b`	`True` if `a` equals `b`
`a != b`	`True` if `a` is not equal to `b`
`a < b`, a <= b	`True` if `a` is less than (less than or equal to) `b`
`a > b, a >= b`	`True` if `a` is greater than (greater than or equal to) `b`
`a is b`	`True` if `a` and `b` reference the same Python object
`a is not b`	`True` if `a` and `b` reference different Python objects

Comparison

- `is` Keyword Usage: To check if two variables reference the same object, use the `is` keyword.
- `is not` for Distinct Objects: Use `is not` to confirm that two objects are not the same.
- Distinction from == Operator: Unlike == which checks for equality in value, `is` checks for identity (i.e., the same object in memory).
- Checking for `None`: Commonly used to check if a variable is `None`, as there is only one instance of `None` in Python.

Comparison

```
[30]: a = [1, 2, 3]

[31]: b = a

[32]: c = list(a)

[33]: a is b

[33]: True

[34]: a is not c

[34]: True

[35]: a == c

[35]: True

[36]: a = None
      a is None

[36]: True
```

Mutable and immutable objects

- Mutable Objects: Lists, dictionaries, NumPy arrays, and most user-defined types in Python are mutable, allowing modification of the object or the values they contain.
- Immutable Objects: Strings and tuples are examples of immutable objects, meaning their internal data cannot be altered after creation.
- Implications of Mutability: Understanding the mutability of objects is crucial for correctly managing data changes and object behaviors in Python programming.

```
[37]: a_list = ["foo", 2, [4, 5]]
      a_list[2] = (3, 4)
      a_list

[37]: ['foo', 2, (3, 4)]

[38]: a_tuple = (3, 5, (4, 5))
      a_tuple[1] = "four"

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[38], line 2
      1 a_tuple = (3, 5, (4, 5))
----> 2 a_tuple[1] = "four"

TypeError: 'tuple' object does not support item assignment
```

Scalar Types

- Scalar Types Overview: Python includes built-in types for handling numerical data, strings, Booleans (True or False), and dates and time, referred to as scalar types.
- Main Scalar Types: The core scalar types include various numeric types (integers, floats), strings, and Booleans.
- Date and Time Handling: Specialized handling of dates and times is managed by the datetime module in Python's standard library.

Type	Description
`None`	The Python "null" value (only one instance of the `None` object exists)
`str`	String type; holds Unicode strings
`bytes`	Raw binary data
`float`	Double-precision floating-point number (note there is no separate `double` type)
`bool`	A Boolean `True` or `False` value
`int`	Arbitrary precision integer

Numeric Types

- Numeric Types - `int` and `float`: Python uses `int` for integers (capable of storing arbitrarily large numbers) and `float` for floating-point numbers (double-precision).
- Floating-Point Representation: Floating-point numbers can be expressed in scientific notation, revealing their underlying double-precision nature.
- Division Operations: Regular division yields a floating-point number, while floor division (//) drops the fractional part, emulating C-style integer division.

Numeric Types

```
[39]: ival = 17239871
      ival ** 6

[39]: 26254519291092456596965462913230729701102721

[40]: fval = 7.243
      fval2 = 6.78e-5

[41]: 3 / 2

[41]: 1.5

[42]: 3 // 2

[42]: 1
```

Strings

- String Literals and Types: Python supports string literals enclosed in single (' ') or double quotes (" "), with `str` being the string type. Multiline strings can be defined using triple quotes (''' ''' or """ """).
- Immutability of Strings: Python strings are immutable, meaning once created, their contents cannot be altered.

```
[43]: c = """
      This is a longer string that
      spans multiple lines
      """

[44]: c.count("\n")

[44]: 3

[45]: a = "this is a string"
      a[10] = "f"

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[45], line 2
      1 a = "this is a string"
----> 2 a[10] = "f"

TypeError: 'str' object does not support item assignment
```

Strings

- Modifying Strings: To alter a string in Python, use functions or methods that create a new string, like the `replace` method of a string object.

```
[46]: b = a.replace("string", "longer string")
      b

[46]: 'this is a longer string'

[47]: a

[47]: 'this is a string'
```

Strings

- String Conversion and Nature: In Python, many objects can be converted to strings using the `str` function. Strings, being sequences of Unicode characters, can be treated similarly to other sequences like lists and tuples.

```
[48]: a = 5.6
      s = str(a)
      print(s)

5.6

[49]: s = "python"
      list(s)
      s[:3]

[49]: 'pyt'
```

Strings

- Escape Character: The backslash (\) in Python is used as an escape character for special sequences like \n (newline), requiring backslashes in strings to be escaped themselves (e.g., \\).
- Raw Strings: Prefixing a string with 'r' (e.g., r\"text") creates a raw string, where backslashes are treated as literal characters and not as escape characters.

```
[50]: s = "12\\34"
      print(s)

      12\34

[51]: s = r"this\has\no\special\characters"
      s

[51]: 'this\\has\\no\\special\\characters'

[52]: a = "this is the first half "
      b = "and this is the second half"
      a + b

[52]: 'this is the first half and this is the second half'
```

Strings

- String Formatting with `format`: Python's string `format` method allows substituting formatted arguments into a string. For instance, `{0:.2f}` formats the first argument as a floating-point number, `{1:s}` as a string, and `{2:d}` as an exact integer.
- f-strings in Python 3.6: Introduced in Python 3.6, f-strings (formatted string literals) simplify string formatting by allowing Python expressions inside curly braces directly within the string literal, prefixed with an 'f' (e.g., `f"Result: {expression}"`).

```
[53]: template = "{0:.2f} {1:s} are worth US${2:d}"

[54]: template.format(88.46, "Argentine Pesos", 1)

[54]: '88.46 Argentine Pesos are worth US$1'

[55]: amount = 10
      rate = 88.46
      currency = "Pesos"
      result = f"{amount} {currency} is worth US${amount / rate}"

[56]: f"{amount} {currency} is worth US${amount / rate:.2f}"

[56]: '10 Pesos is worth US$0.11'
```

Bytes and Unicode

- Unicode in Modern Python: From Python 3.0 onwards, Unicode is the primary string type, facilitating consistent handling of both ASCII and non-ASCII text.
- Unicode vs. Bytes in Older Python: In older Python versions, strings were bytes without explicit Unicode encoding, requiring manual conversion to Unicode if the character encoding was known.

```
[57]: val = "español"
      val

[57]: 'español'

[58]: val_utf8 = val.encode("utf-8")
      val_utf8
      type(val_utf8)

[58]: bytes

[59]: val_utf8.decode("utf-8")

[59]: 'español'

[60]: val.encode("latin1")
      val.encode("utf-16")
```

Boolean

- Boolean Values and Operations: In Python, the two Boolean values are written as `True` and `False`, used in comparisons and conditional expressions, and can be combined with `and` and `or` keywords.
- Conversions and the `not` Keyword: `False` converts to 0 and `True` to 1 when cast to numbers, while `not` inverts a Boolean value (`True` to `False` or vice versa).

Boolean

```
[62]: True and True

[62]: True

[63]: False or True

[63]: True

[66]: int(False)

[66]: 0

[67]: int(True)

[67]: 1

[68]: a = True
      b = False
      print(not a)
      not b

      False
[68]: True
```

Dates and times

- Python's `datetime` Module: Provides `datetime`, `date`, and `time` types, with `datetime` being the most commonly used as it combines both date and time information.
- Extracting Date and Time: From a `datetime` instance, the equivalent `date` and `time` objects can be extracted using methods named after the types themselves.

```
[71]: from datetime import datetime, date, time
      dt = datetime(2011, 10, 29, 20, 30, 21)

[72]: dt.day

[72]: 29

[73]: dt.minute

[73]: 30

[75]: dt.date()

[75]: datetime.date(2011, 10, 29)

[76]: dt.time()

[76]: datetime.time(20, 30, 21)
```

Dates and times

- Formatting and Parsing: The `strftime` method is used to format a datetime object as a string, while `strptime` converts strings into datetime objects.
- Modifying Datetime Objects: Methods like `replace` can alter specific time fields in datetime objects, producing new instances as `datetime.datetime` is immutable (e.g., setting minute and second to zero).

Dates and times

```
[77]: dt.strftime("%Y-%m-%d %H:%M")

[77]: '2011-10-29 20:30'

[78]: datetime.strptime("20091031", "%Y%m%d")

[78]: datetime.datetime(2009, 10, 31, 0, 0)

[79]: dt_hour = dt.replace(minute=0, second=0)
      dt_hour

[79]: datetime.datetime(2011, 10, 29, 20, 0)

[80]: dt

[80]: datetime.datetime(2011, 10, 29, 20, 30, 21)

[81]: dt2 = datetime(2011, 11, 15, 22, 30)
      delta = dt2 - dt
      delta
      type(delta)

[81]: datetime.timedelta
```

Control Flow

- Standard Control Flow Keywords: Python includes built-in keywords for implementing conditional logic, loops, and other control flow constructs commonly found in programming languages.
- Usage of `if`, `elif`, and `else`: The `if` statement evaluates a condition and executes the subsequent code block if the condition is `True`, with `elif` and `else` providing additional conditional and alternative execution paths.

```
[83]: a = 5; b = 7
      c = 8; d = 4
      if a < b or c > d:
          print("Made it")

      Made it

[84]: 4 > 3 > 2 > 1

[84]: True
```

SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Spring 2024