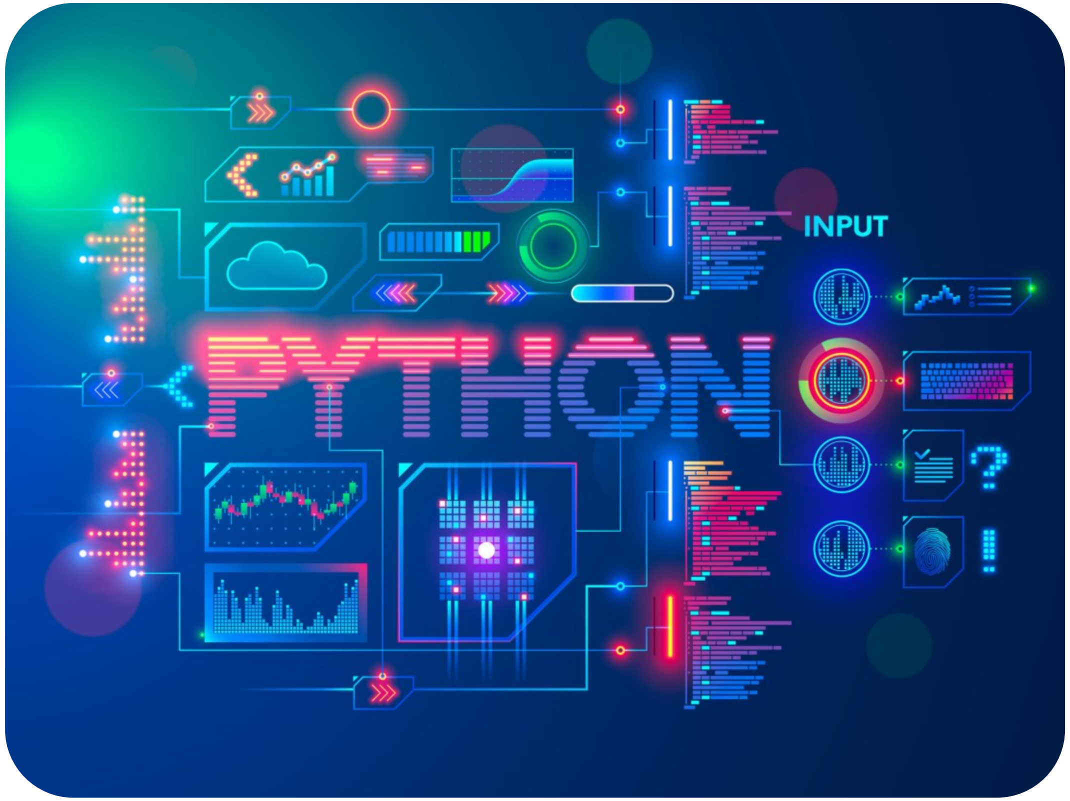
COLUMBUS STATE
COMMUNITY COLLEGE



WIIT: 7740
Scripting with Python

Week 1: Python Strings and `Print()`

Machine & High-Level Programming Comparison

Machine (Low-Level) Language vs High-Level Programming

Comparison	Low-Level	High-Level Programming
Basic	Machine-Friendly	Programmer-Friendly
Memory Efficiency	High	Low
Execution	Fast	Slow
Portability and Machine Dependency	Non-portable and machine **DEPENDENT**	Portable and machine **INDEPENDENT**; can be run on **ANY** platform.
Translation	Assembler is required while machine language is directly executed	Requires compiler or an interpreter
Debugging and Maintenance	Complex	Simple

Procedure-Oriented & Object-Oriented Programming Comparison

Procedure-Oriented vs. Object-Oriented Programming

Procedure-Oriented	Object-Oriented
Program is divided into small parts called **functions**.	Program is divided into small parts called **objects**.
follows **top-down approach**.	follows **bottom-up approach**.
Adding new data/function: **Difficult**	Adding new data/function: **Easy**
Less Secure	More secure
Functionality is MORE important than data	Data is MORE important than functionality

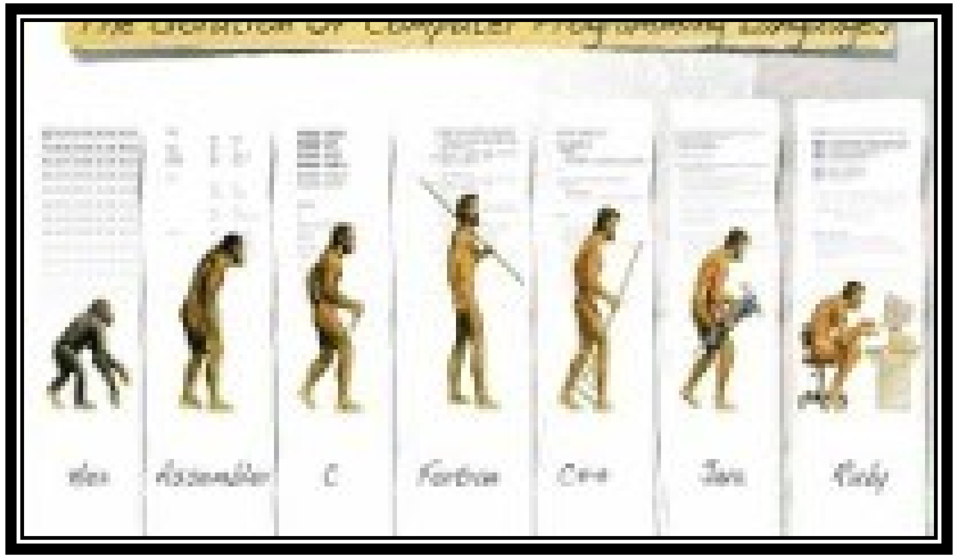
Brief History

Key points

- Programming was recognized clear back to the mid-1800’s
- 1940’s saw the beginning of concentrated efforts

Early Languages

- 1948 – first computer program run
- 1949 Short Code
- Early 1950’s Autocode
- 1954 FORTRAN
- 1959 COBOL (Still widely in use today)



About Python

- 1989 Python initiated
- Guido van Rossum - creator of Python
- Roots in the programming language ABC (which he also helped develop)
- Named after Monty Python’s Flying Circus
- Does not require compiling

Idiosyncrasies:

- Case Sensitive
- Tabs and Spaces have meaning
- Ends of lines are inferred (versus explicit)

Expectations Of Your Code

DOCUMENTATION

- Docstrings

```
"""
FILE_NAME
AUTHOR
DATE
PURPOSE
"""
```

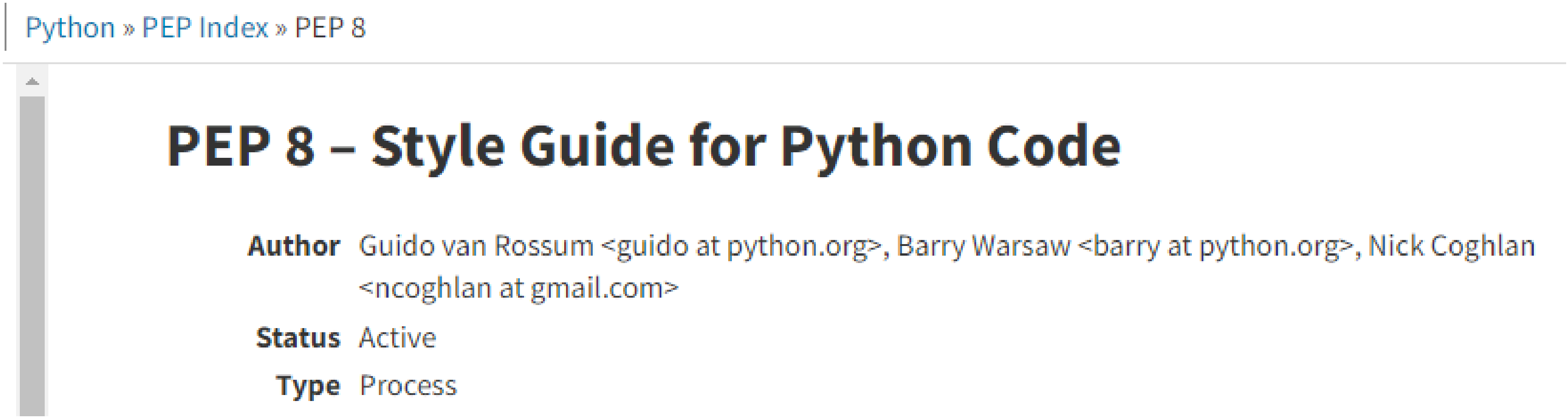
- Commenting (#)
- Smart Variable Naming
- Appropriate Indentations (important!)

```
        self.file = None
        self.fingerprints = set()
        self.logdupes = True
        self.debug = debug
        self.logger = logging.getLogger(__name__)
        if path:
            self.file = open(os.path.join(path, 'requests.seen'), 'a+')
            self.file.seek(0)
            self.fingerprints.update(x.rstrip() for x in self.file)

    @classmethod
    def from_settings(cls, settings):
        debug = settings.getbool('DUPEFILTER_DEBUG')
        return cls(job_dir(settings), debug)

    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        if fp in self.fingerprints:
            return True
        self.fingerprints.add(fp)
        if self.file:
            self.file.write(fp + os.linesep)

    def request_fingerprint(self, request):
        return request_fingerprint(request)
```

Expectations Of Your Code: PEP-8 and Style Guide

Style Guide:

https://www.python.org/dev/peps/pep-0008/



Python » PEP Index » PEP 8

PEP 8 – Style Guide for Python Code

Author Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>

Status Active

Type Process

HELPFUL TIPS

- **Experiment** and **Intentionally** design mistakes into your code to see what happens.
- Run code **EARLY** and **OFTEN**
- Do **ONE** hard thing at a time, “break down” the problem into different pieces
- Use Pseudocode to write the program in plain English
- Create flow diagrams to understand logical steps
  - https://www.lucidchart.com/
- Use the **Debugger** to find the source of functional problems

Today’s Coding Goals

- See a working program in action
- Practice writing code yourself

Learn Concepts:

- **`print()`**
- **`type()`**
- Strings

Capabilities of Python

- Create **data** (number or string of characters) and **label** it (for later reference)
  - This is called a 'variable'
- **Output** data to the user
- **Input** data from the user
- **Calculate** and transform new data from old data (e.g. arithmetic)
- **Convert** data from one type to another

print()

- Built in Python function that prints the specified data to the standard output device (in most cases your command terminal)
- Anything can be passed to **`print`** as an argument, but what can be expected to print can vary:

```
print('Hello World!')
```

Output:

```
Hello World!
```

```
print(int)
```

Output:

```
<class 'int'>
```

type()

Common types:

`str, float, int, list, dict, tuple`

- Returns the Class type of the object passed to the method as an argument
- Types can be validated using the **`is`** keyword which checks if the types on either side are the same

```
print(type('my_string') is str)
```

Output:

```
True
```

```
print(type(123) is int)
```

Output:

```
True
```

```
print(type('my_string')) is int)
```

Output:

```
False
```

```
print(type(123)) is str)
```

Output:

```
False
```

Strings

Key Points:

- Strings are sequences of characters. Items in these sequences can be accessed using 'indices' or 'slices'.
  - index: `'my_string'[4]    == 't'`
  - slice: `'my_string'[1:5] == 'y_st'`
- Indices in Python start at 0, but negative indices can be used to select an index from the opposite end.
  - Example: [-1] will always access the last index in a string
- Strings themselves cannot be modified, but they can be transformed and produce new strings
- You can identify length, min and max values, first and last values
- You can iterate over strings using loops (will be covered later)

Strings continued

Key Points:

- Strings are sequences of characters. Items in these sequences can be accessed using 'indices' or 'slices'.
  - index: `'my_string'[4]   == 't'`
  - slice: `'my_string'[1:5] == 'y_st'`
- Indices in Python start at 0, but negative indices can be used to select an index from the opposite end.
  - Example: [-1] will always access the last index in a string
- Strings themselves cannot be modified, but they can be transformed and produce new strings
- You can identify length, min and max values, first and last values
- You can iterate over strings using loops (will be covered later)

Glossary: Types, Functions & Operators

Types

`int`

Numeric type for integers

`float`

Numeric type for decimal numbers

`string`

Text type containing sequences of characters

Functions

`type(object)`

returns the class type of the argument(object)

`print(object(s), sep, end, file, flush)`

prints object(s) to the standard output

Operators

=	assignment operator; assign values to variables
+	addition
-	subtraction
*	multiplication
/	division

Glossary: Terminology

Terms

object

an instance of a class in Python that contains data and functions to manipulate that data

variable

container for storing data values to be used later in the program; the data can be objects

iterable

an object containing a sequence of other objects that can be iterated over (e.g., going down a grocery list and checking things off; the shopping list is iterable)

index

location of an item inside an iterable object

slice

sub-section of a sequence; for strings, this is a sequence of characters between two indices.

Columbus State
Community College

Practice Coding:
Class Activity