

Programming Principles in Python (CSCI 503/490)

Regular Expressions & Files

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Unicode and ASCII

- Conceptual systems
- ASCII:
 - old, English-centric, 7-bit system (only 128 characters)
- Unicode:
 - Can represent over 1 million characters from all languages + emoji 🎉
 - Characters have hexadecimal representation: é = U+00E9 and name (LATIN SMALL LETTER E WITH ACUTE)
 - Python allows you to type "é" or represent via code "\u00e9"
- Codes: `ord` → character to integer, `chr` → integer to character

Strings

- Objects with methods
- Finding and counting substrings: `count`, `find`, `startswith`
- Removing leading & trailing substrings/whitespace: `strip`, `removeprefix`
- Transforming Text: `replace`, `upper`, `lower`, `title`
- Checking String Composition: `isalnum`, `isnumeric`, `isupper`

Splitting & Joining

- `s = "Venkata, Ranjit, Pankaj, Ali, Karthika"`
- Split:
 - `names = s.split(',') # names is a list`
 - `names = s.split(',', 3) # split by commas, split <= 3 times`
 - `rsplit` works in reverse, from the right of the string
- Join
 - Inverse of split, but an instance function of the **separator string**
 - `','.join(names)`

Joining

- `join` is a method on the **separator** used to join a list of strings
- `' , '.join(names)`
 - `names` is a list of strings, `' , '` is the separator used to join them
- Example:
 - ```
def orbit(n):
 # ...
 return orbit_as_list
print(' , '.join(orbit_as_list))
```

# Format and f-Strings

---

- `s.format`: templating function
  - Replace fields indicated by curly braces with corresponding values
  - `"My name is {} {}".format(first_name, last_name)`
  - `"My name is {first_name} {last_name}".format(  
first_name=name[0], last_name=name[1])`
- Formatted string literals (f-strings) reference variables **directly!**
  - `f"My name is {first_name} {last_name}"`
- Can include expressions, too:
  - `f"My name is {name[0].capitalize()} {name[1].capitalize()}"`
- Format mini-language allows specialized displays (alignment, numeric formatting)

# CSAN Panel: Real Jobs in the Real World



**NIU**

COMPUTER SCIENCE  
ALUMNI NETWORK

**REAL  
JOBS IN  
THE REAL  
WORLD**

A Panel  
Discussion

**TUESDAY, OCT. 7, 2025**  
Barsema Alumni & Visitors Center (Ballroom)  
5:30–7:30 p.m.

- Tuesday, Oct. 7, 5:30–7:30pm
- Provides an insight into jobs from NIU alumni
- Food is Provided
- Sponsored by the Computer Science Alumni Network and the NIU Alumni Association

# Assignment 4

---

- Assignment covers strings and files
- Reading & writing data to files
- Deals with characters and formatting

# Raw Strings

---

- Raw strings prefix the starting delimiter with `r`
- Disallow escaped characters
- `'\\n` is the way you write a newline, `\\\\` for `\\`.
- `r"\n` is the way you write a newline, `\\` for `\`.
- Useful for regular expressions

# Regular Expressions

---

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
- "31" in "The last day of December is 12/31/2016."
- May work for some questions but now suppose I have other lines like: "The last day of September is 9/30/2016."
- ...and I want to find dates that look like:
- {digits}/{digits}/{digits}
- Cannot search for every combination!
- \d+/\d+/\d+ # \d is a **character class**

# Metacharacters

---

- Need to have some syntax to indicate things like repeat or one-of-these or this is optional.
- . ^ \$ \* + ? { } [ ] \ | ( )
- [ ]: define character class
- ^: complement (opposite)
- \: escape, but now escapes metacharacters and references classes
- \*: repeat zero or more times
- +: repeat one or more times
- ?: zero or one time
- {m, n}: at least m and at most n

# Predefined Character Classes

---

| Character class | Matches                                                                   |
|-----------------|---------------------------------------------------------------------------|
| <code>\d</code> | Any digit (0–9).                                                          |
| <code>\D</code> | Any character that is <i>not</i> a digit.                                 |
| <code>\s</code> | Any whitespace character (such as spaces, tabs and newlines).             |
| <code>\S</code> | Any character that is <i>not</i> a whitespace character.                  |
| <code>\w</code> | Any <b>word character</b> (also called an <b>alphanumeric character</b> ) |
| <code>\W</code> | Any character that is <i>not</i> a word character.                        |

[Deitel & Deitel]

# Performing Matches

---

| Method/Attribute        | Purpose                                                                                     |
|-------------------------|---------------------------------------------------------------------------------------------|
| <code>match()</code>    | Determine if the RE matches at the beginning of the string.                                 |
| <code>search()</code>   | Scan through a string, looking for any location where this RE matches.                      |
| <code>findall()</code>  | Find all substrings where the RE matches, and returns them as a list.                       |
| <code>finditer()</code> | Find all substrings where the RE matches, and returns them as an <a href="#">iterator</a> . |

# Regular Expressions in Python

---

- `import re`
- `re.match(<pattern>, <str_to_check>)`
  - Returns `None` if no match, information about the match otherwise
  - Starts at the **beginning** of the string
- `re.search(<pattern>, <str_to_check>)`
  - Finds **single** match **anywhere** in the string
- `re.findall(<pattern>, <str_to_check>)`
  - Finds **all** matches in the string, `search` only finds the first match
- Can pass in flags to alter methods: e.g. `re.IGNORECASE`

# Examples

---

- `s0 = "No full dates here, just 02/15"`  
  `s1 = "02/14/2021 is a date"`  
  `s2 = "Another date is 12/25/2020"`
- `re.match(r'\d+/\d+/\d+', s1)` # returns match object
- `re.match(r'\d+/\d+/\d+', s0)` # None
- `re.match(r'\d+/\d+/\d+', s2)` # None!
- `re.search(r'\d+/\d+/\d+', s2)` # returns 1 match object
- `re.search(r'\d+/\d+/\d+', s3)` # returns 1! match object
- `re.findall(r'\d+/\d+/\d+', s3)` # returns list of strings
- `re.finditer(r'\d+/\d+/\d+', s3)` # returns iterable of matches

# Grouping

---

- Parentheses capture a group that can be accessed or used later
- Access via `groups()` or `group(n)` where `n` is the number of the group, but numbering starts at **1**
- Note: `group(0)` is the **full** matched string
- ```
for match in re.finditer(r'(\d+)/(\d+)/(\d+)', s3):  
    print(match.groups())
```
- ```
for match in re.finditer(r'(\d+)/(\d+)/(\d+)', s3):
 print('{2}-{0:02d}-{1:02d}'.format(
 *[int(x) for x in match.groups()]))
```
- `*` operator expands a list into individual elements

# Modifying Strings

---

| Method/Attribute     | Purpose                                                                                               |
|----------------------|-------------------------------------------------------------------------------------------------------|
| <code>split()</code> | Split the string into a list, splitting it wherever the RE matches                                    |
| <code>sub()</code>   | Find all substrings where the RE matches, and replace them with a different string                    |
| <code>subn()</code>  | Does the same thing as <code>sub()</code> , but returns the new string and the number of replacements |

# Substitution

---

- Do substitution in the middle of a string:
- `re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', s3)`
- All matches are substituted
- First argument is the regular expression to **match**
- Second argument is the **substitution**
  - `\1`, `\2`, ... match up to the **captured groups** in the first argument
- Third argument is the **string** to perform substitution on
- Can also use a **function**:
- `to_date = lambda m:`  
`f'{m.group(3)}-{int(m.group(1)):02d}-{int(m.group(2)):02d}'`  
`re.sub(r'(\d+)/(\d+)/(\d+)', to_date, s3)`

# Files

# Files

---

- A file is a sequence of data stored on disk.
- Python uses the standard Unix newline character (`\n`) to mark line breaks.
  - On Windows, end of line is marked by `\r\n`, i.e., carriage return + newline.
  - On old Macs, it was carriage return `\r` only.
  - Python **converts** these to `\n` when reading.

# Opening a File

---

- Opening associates a file on disk with an object in memory (file object or file handle).
- We access the file via the **file object**.
- `<filevar> = open(<name>, <mode>)`
- Mode `'r'` = read or `'w'` = write, `'a'` = append
- read is default
- Also add `'b'` to indicate the file should be opened in binary mode: `'rb'`, `'wb'`

# Standard File Objects

---

- When Python begins, it associates three standard file objects:
  - `sys.stdin`: for input
  - `sys.stdout`: for output
  - `sys.stderr`: for errors
- In the notebook
  - `sys.stdin` isn't really used, `get_input` can be used if necessary
  - `sys.stdout` is the output shown after the code
  - `sys.stderr` is shown with a red background

# Files and Jupyter

---

- You can **double-click** a file to see its contents (and edit it manually)
- To see one as text, may need to right-click
- **Shell commands** also help show files in the notebook
- The `!` character indicates a shell command is being called
- These will work for Linux and macos but not necessarily for Windows
- `!cat <fname>`: print the entire contents of `<fname>`
- `!head -n <num> <fname>`: print the first `<num>` lines of `<fname>`
- `!tail -n <num> <fname>`: print the last `<num>` lines of `<fname>`

# Reading Files

---

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate read (default)
- Can iterate through the file (think of the file as a collection of lines):
  - ```
f = open('huck-finn.txt', 'r')
for line in f:
    if 'Huckleberry' in line:
        print(line.strip())
```
- Using `line.strip()` because the read includes the newline, and `print` writes a newline so we would have double-spaced text
- Closing the file: `f.close()`

Remember Encodings (Unicode, ASCII)?

- Encoding: How things are actually stored
- ASCII "Extensions": how to represent characters for different languages
 - No universal extension for 256 characters (one byte), so...
 - ISO-8859-1, ISO-8859-2, CP-1252, etc.
- Unicode encoding:
 - UTF-8: used in Python and elsewhere (uses variable # of 1 — 4 bytes)
 - Also UTF-16 (2 or 4 bytes) and UTF-32 (4 bytes for everything)
 - Byte Order Mark (BOM) for files to indicate endianness (which byte first)

Encoding in Files

- `all_lines = open('huck-finn.txt').readlines()`
`all_lines[0] # '\ufeff\n'`
- `\ufeff` is the UTF Byte-Order-Mark (BOM)
- Optional for UTF-8, but if added, need to read it
- `a = open('huck-finn.txt', encoding='utf-8-sig').readlines()`
`a[0] # '\n'`
- No need to specify UTF-8 (or ascii since it is a subset)
- Other possible encodings:
 - cp1252, utf-16, iso-8859-1

Other Methods for Reading Files

- `read()`: read the entire file
- `read(<num>)`: read <num> characters (bytes)
 - `open('huck-finn.txt', encoding='utf-8-sig').read(100)`
- `readlines()`: read the entire file as a list of lines
 - `lines = open('huck-finn.txt', encoding='utf-8-sig').readlines()`