# Programming Principles in Python (CSCI 503/490)

## Strings

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

# Generators

- Special functions that return **lazy** iterables
- Use less memory
- Change is that functions `yield` instead of `return`
- ```
  def square(it):
      for i in it:
          yield i*i
  ```
- If we are iterating through a generator, we hit the first yield and immediately return that first computation
- Generator expressions just shorthand (remember no tuple comprehensions)
  - `(i * i for i in [1,2,3,4,5])`
  ```

# Efficient Evaluation

- Only compute when necessary, not beforehand
- ~~`u = compute_fast_function(s, t)`~~
  ~~`v = compute_slow_function(s, t)`~~

```
if s > t and s**2 + t**2 > 100:
    u = compute_fast_function(s, t)
    res = u / 100
else:
    v = compute_slow_function(s, t)
    res = v / 100
```

- slow function will not be executed unless the condition is true

# Short-Circuit Evaluation

- Automatic, works left to right according to order of operations (and before or)
- Works for `and` and `or`
- and:
  - if **any** value is `False`, stop and return `False`
  - 
```
a, b = 2, 3
a > 3 and b < 5
```
- or:
  - if **any** value is `True`, stop and return `True`
  - 
```
a, b, c = 2, 3, 7
a > 3 or b < 5 or c > 8
```

# Memoization

- ```
  memo_dict = {}
  def memoized_slow_function(s, t):
      if (s, t) not in memo_dict:
          memo_dict[(s, t)] = compute_slow_function(s, t)
      return memo_dict[(s, t)]
  ```
- ```
  for s, t in [(12, 10), (4, 5), (5, 4), (12, 10)]:
      if s > t and (c := memoized_slow_function(s, t) > 50):
          pass
      else:
          c = compute_fast_function(s, t)
  ```
- Second time executing for `s=12`, `t=10`, we don't need to compute!
- Tradeoff memory for compute time

# Functional Programming

- Programming without imperative statements like assignment
- In addition to comprehensions & iterators, have functions:
  - map: iterable of n values to an iterable of n transformed values
  - filter: iterable of n values to an iterable of m (m <= n) values
- Eliminates need for concrete looping constructs

# Lambda Functions

- `def is_even(x):`
  `    return (x % 2) == 0`
- `filter(is_even, range(10) # generator`
- Lots of code to write a simple check
- Lambda functions allow inline function definition
- Usually used for "one-liners": a simple data transform/expression
- `filter(lambda x: x % 2 == 0, range(10))`
- Parameters follow `lambda`, **no parentheses**
- **No** `return` keyword as this is implicit in the syntax
- JavaScript has similar functionality (arrow functions): `(d => d % 2 == 0)`

# Assignment 3

- Important for Test 1, but studying also should be a priority
- Deadline moved to Friday, Feb. 24
- Pokémon Data
- Looking at where and how people and goods move across land borders
- Start with the sample notebook (or copy its code) to download the data
- Data is a list of dictionaries
- Need to iterate through, update, and create new lists & dictionaries

# Test 1

- This Wednesday, Feb. 22, 11:00am-12:15pm
- In-Class, paper/pen & pencil
- Covers material through last week
- Format:
  - Multiple Choice
  - Free Response
- Information at the link above

# Remote Office Hours Today

- Due to family illness, need to conduct office hours remotely today (Zoom)
- Please email me with questions or for appointments

# Strings

- Remember strings are sequences of characters
- Strings are collections so have `len`, `in`, and iteration
  - `s = "Huskies"`
    `len(s); "usk" in s; [c for c in s if c == 's']`
- Strings are sequences so have
  - indexing and slicing: `s[0], s[1:]`
  - concatenation and repetition: `s + " at NIU"; s * 2`
- Single or double quotes `'string1'`, `"string2"`
- Triple double-quotes: `"""A string over many lines"""`
- Escaped characters: `'\n'` (newline) `'\t'` (tab)

# Unicode and ASCII

- Conceptual systems
- ASCII:
  - old 7-bit system (only 128 characters)
  - English-centric
- Unicode:
  - modern system
  - 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"`

# Unicode and 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)

# Codes

- Characters are still stored as bits and thus can be represented by numbers
  - `ord` → character to integer
  - `chr` → integer to character
  - `"\N{horse}"`: named emoji

# Strings are Objects with Methods

- We can call methods on strings like we can with lists
  - 
```
s = "Peter Piper picked a peck of pickled peppers"
s.count('p')
```
- Doesn't matter if we have a variable or a literal
  - 
```
"Peter Piper picked a peck of pickled peppers".find("pick")
```

# Finding & Counting Substrings

- `s.count(sub)`: Count the number of occurrences of `sub` in `s`
- `s.find(sub)`: Find the first position where `sub` occurs in `s`, else `-1`
- `s.rfind(sub)`: Like `find`, but returns the right-most position
- `s.index(sub)`: Like `find`, but raises a ValueError if not found
- `s.rindex(sub)`: Like `index`, but returns right-most position
- `sub in s`: Returns `True` if `s` contains `sub`
- `s.startswith(sub)`: Returns `True` if `s` starts with `sub`
- `s.endswith(sub)`: Returns `True` if `s` ends with `sub`

# Removing Leading and Trailing Strings

- `s.strip()`: Copy of `s` with leading and trailing whitespace removed
- `s.lstrip()`: Copy of `s` with leading whitespace removed
- `s.rstrip()`: Copy of `s` with trailing whitespace removed
- `s.removeprefix(prefix)`: Copy of `s` with `prefix` removed (if it exists)
- `s.removesuffix(suffix)`: Copy of `s` with `suffix` removed (if it exists)

# Transforming Text

- `s.replace(oldsub, newsub)`:
  Copy of `s` with occurrences of `oldsub` in `s` with `newsub`
- `s.upper()`: Copy of `s` with all uppercase characters
- `s.lower()`: Copy of `s` with all lowercase characters
- `s.capitalize()`: Copy of `s` with first character capitalized
- `s.title()`: Copy of `s` with first character of each word capitalized

# Checking String Composition

| String Method | Description |
|---|---|
| `isalnum()` | Returns True if the string contains only alphanumeric characters (i.e., digits & letters). |
| `isalpha()` | Returns True if the string contains only alphabetic characters (i.e., letters). |
| `isdecimal()` | Returns True if the string contains only decimal integer characters |
| `isdigit()` | Returns True if the string contains only digits (e.g., '0', '1', '2'). |
| `isidentifier()` | Returns True if the string represents a valid identifier. |
| `islower()` | Returns True if all alphabetic characters in the string are lowercase characters |
| `isnumeric()` | Returns True if the characters in the string represent a numeric value w/o a + or - or . |
| `isspace()` | Returns True if the string contains only whitespace characters. |
| `istitle()` | Returns True if the first character of each word is the only uppercase character in it. |
| `isupper()` | Returns True if all alphabetic characters in the string are uppercase characters |

[Deitel & Deitel]

# Splitting

- `s = "Venkata, Ranjit, Pankaj, Ali, Karthika"`
- `names = s.split(',') # names is a list`
- `names = s.split(',', 3) # split by commas, split <= 3 times`
- separator may be multiple characters
- if no separator is supplied (`sep=None`), runs of consecutive whitespace delimit elements
- `rsplit` works in reverse, from the right of the string
- `partition` and `rpartition` for a single split with before, sep, and after
- `splitlines` splits at line boundaries, optional parameter to keep endings

# 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))
    ```
    ```

# Formatting

- `s.ljust`, `s.rjust`: justify strings by adding fill characters to obtain a string with specified width
- `s.zfill`: `ljust` with zeroes
- `s.format`: templating function
  - Replace fields indicated by curly braces with corresponding values
  - `"My name is {} {}".format(first_name, last_name)`
  - `"My name is {1} {0}".format(last_name, first_name)`
  - 
```
"My name is {first_name} {last_name}.format(
                        first_name=name[0], last_name=name[1])
```
  - Braces can contain number or name of keyword argument
  - Whole format mini-language to control formatting

# Format Strings

- Formatted string literals (f-strings) prefix the starting delimiter with `f`
- 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()}"`
- Same format mini-language is available

# Format Mini-Language Presentation Types

- Not usually required for obvious types
- `:d` for integers
- `:c` for characters
- `:s` for strings
- `:e` or `:f` for floating point
  - `e`: scientific notation (all but one digit after decimal point)
  - `f`: fixed-point notation (decimal number)

# Field Widths and Alignments

- After : but before presentation type
  - `f'[{27:10d}]' # '[        27]'`
  - `f'[{"hello":10}]' # '[hello     ]'`
- Shift alignment using < or >:
  - `f'[{"hello":>15}]' # '[          hello]'`
- Center align using ^:
  - `f'[{"hello":^7}]' # '[ hello ]'`

# Numeric Formatting

- Add positive sign:
  - `f'[{27:+10d}]' # '[       +27]'`
- Add space but only show negative numbers:
  - `print(f'{27: d}\n{-27: d}') # note the space in front of 27`
- Separators:
  - `f'{12345678:,d}' # '12,345,678'`

# 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 |
|---|---|
| `\d` | Any digit (0–9). |
| `\D` | Any character that is *not* a digit. |
| `\s` | Any whitespace character (such as spaces, tabs and newlines). |
| `\S` | Any character that is *not* a whitespace character. |
| `\w` | Any **word character** (also called an **alphanumeric character**) |
| `\W` | Any character that is *not* a word character. |

[Deitel & Deitel]

# Performing Matches

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

# 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`
  ```