- is `python` not working for you in the Git Bash command line in windows?
    - see: https://stackoverflow.com/questions/22869192/git-bash-wont-run-my-python-files
- go over set of exercises from class 1
- `for i in range(5)` is identical to `for i in [0, 1, 2, 3, 4]`, just more compact

⟩ Python basics 2

- **IPython** as replacement for plain Python interpreter

    - IPython is an improved, more interactive version of the plain Python interpreter
    - run `ipython` instead of `python` at the command line
    - does syntax highlighting - pretty colours!
    - numbered input and output lines
        - can access previous output and input lines numbered `N` with `_N` and `_iN`
    - use `?` for help, if available, e.g. `range?` gives help on the `range()` function
        - if help is long, scroll up/down with arrow keys, to exit hit `q`
    - command/variable name completion:
        - `something = 1`
        - `som + <TAB> -> something`
    - command history with up/down keys (same as in plain Python)
    - attribute exploration via dot notation:
        - `s = 'hello'`
        - `s. + <TAB>` gives a popup menu of attributes/methods
    - view all local variables with `whos`
        - you can delete a variable `v` with `del v` (also in plain Python)
        - clear all variables with `reset`
    - paste multiline code from editor directly into IPython
    - automatically indents 4 spaces when appropriate, e.g. when writing a `for` loop
    - run a script using `run`, e.g. `run hellos.py`
        - if you want your script to be able to access existing variables in your workspace, run "interactively" with `run -i hellos.py`
    - can call some bash-like commands for folder navigation: `pwd`, `ls`, `cd`
        - this works even on systems without a bash command line installed
    - exiting **plain Python** in Linux/Mac: `Ctrl+D`; in Windows: `Ctrl+Z`, then ENTER
    - exiting **IPython** in all OSes: `CTRL+D`, or type `exit` or `quit`

- plain text editors

    - key features:
        - plain text format: `.txt`, `.py`, etc.
        - fixed-width font
        - syntax highlighting
        - line numbering
    - linux: geany, gedit, mousepad
    - windows: geany, notepad, notepad++, ultraedit, textpad
    - mac: geany, textedit, atom, sublime, xcode
    - alternatively, can use a cross-platform Python integrated development environments (IDE): pycharm, spyder, JupyterLab
        - downside: bigger, slower, more complicated than simple text editor

⟩ strings

- string operators:

    - initialize a blank string: `s = ''` (can also use double quotes `"` , but single are easier)
    - combine strings with `+` : `s = 'Hello' + ' ' + 'world'`
    - append to an existing string with `+=` : `s += '!'`
    - duplicate strings with `*` : `ss = s * 2`
    - whitespace characters: `\n` (new line) and `\t` (tab, rarely used)
        - to see what they do: `print('Hi\nthere!\n')` , `print('Hi\tthere\t!')`
    - `%` string formatting operator
        - `thing = 'world'`
        - `'Hello %s' % thing`
        - `%s` is a "format string", `%` is the "string formatting" operator
        - format strings act as placeholders, to be replaced by whatever follows the string formatting operator:
        - `%s` - format as a string
        - `%d` - format as a (discrete) integer
            - `'The year %d is here' % 2021`
        - `%f` - format as a float
            - `'pi is %f' % 3.14159`
            - `%.3f` - keep only the first 3 decimal places, round appropriately
            - `'pi is %.3f' % 3.14159`
        - `%g` - (general) format, using only as many digits as necessary (up to a default max of 6)
            - `'I am %g years old' % 10`
            - `'I am %g years old' % 10.5`
            - `'I am %g years old' % 10.56789`
            - `'I am %.7g years old' % 10.56789` - increases the max to 7 digits
        - can replace multiple placeholders in a string at once, using a "tuple" `()`
            - `'The date is %s %d, %d' % ('April', 28, 2020)`
        - what else does `%` do in Python?
            - how does Python know whether to use `%` as a string formatting operator or as mod operator?

- example string: `s = 'abcdefg'`

    - get length by calling the `len()` function: `len(s)` gives `7`
    - check if a string exists within another string using `in` : `'h' in s` gives `False` , `'cde' in s` gives `True`
        - where have we seen the `in` operator before?
    - can iterate over the characters in a string, also using `in` :

    ```
    for c in s:
        print(c)
    ```

    - **indexing** lets you extract a single entry:
        - `s[0]` gives `'a'` , `s[1]` gives `'b'` , etc.
        - this is called "0-based" indexing, similar in behaviour to `range()`
        - 0-based indexing is used throughout Python (Matlab is 1-based)

- negative index counts from the end: `s[-1]` gives `'g'` , `s[-2]` gives `'f'` , etc.
  - **slicing** lets you extract multiple entries at once:
    - `s[0:1]` gives `'a'` , `s[0:2]` gives `'ab'` , `s[1:3]` gives `'bc'` , etc.
    - slice indices are like fenceposts, they retrieve fence segments that fall in between them
    - normal (non-slice) indices used for normal indexing give you the fence segments directly
    - you can also skip over entries when slicing
    - `s[0:7:2]` -> `aceg` - give me all the entries from fencepost 0 to 7 in steps of 2
    - `s[0:7:3]` -> `adg`
    - if you leave out a slice index, its value is implied:
      - leave out the first slice index -> start from beginning of string - `s[:7:2]`
      - leave out the 2nd slice index: go to end of string - `s[0::2]`
      - leave out the 3rd slice index: go in steps of 1 - `s[0:7]` or `s[0:7:]`
      - leave out multiple slice indices: `s[::2]` - start to end, steps of 2
      - reverse a string using a negative slice index: `s[::-1]` - end to start, steps of 1

- string methods

  - everything in Python is an "object", `type()` tells you what kind of object it is
  - objects can have "attributes", which are like adjectives
    - `n = 1.0`
    - `n.<TAB>` in IPython to discover all attributes and methods of an object
    - `n.real` - real part of `n`
    - `n.imag` - imaginary part of `n`
    - compare with complex number `n = 1+2j`
  - objects can also have "methods" (denoted by `()` in `.<TAB>` expansion)
    - methods are functions that only apply to that object
    - methods are like verbs
  - like a normal function (e.g. `print()`), methods take inputs and return outputs
  - `s = 'hello world!'` , `a = 'o'`
  - `s.count(a)` - find number of occurences of `a` in `s`
  - `s.index(a)` - find 0-based index (position) of first instance of string `a` in `s`
  - `s.split(a)` - split `s` everywhere that string `a` is found
  - `s.join(a, b, c)` - join strings a, b, c... separating them by the string in `s`
  - `s.replace(old, new)` - find all instances of string `old` , replace with `new`
  - `s.strip(a)` - strip characters in `a` from start and end of `s` , defaults to stripping spaces
    - what might `s.lstrip()` and `s.rstrip()` do?
  - `s.upper()` - convert `s` to uppercase
  - `s.lower()` - convert `s` to lowercase
  - what would `s.upper().lower()` do?
    - can chain multiple methods together iff method1 returns an object with a method2
  - to discover what each method does without having to do a web search:
    - in IPython, `s.` + `<TAB>` , select the method, then type `?` for help

> **string exercises**

1. Store the alphabet `abcdefghijklmnopqrstuvwxyz` in a string `s`. Use a `for` loop to print out the alphabet backwards. Now do the same thing in a single line of code, in a single line of output
2. Collect every 2nd letter in the alphabet, and store them all together in a single string
3. Make a new string that takes the above string and replaces `'a'` with `'4'`, `'e'` with `'3'`, and `'i'` with `'1'`

› *defining* **your own functions**

- function: takes inputs, returns output(s)
- function inputs are called "arguments"

```
def add(x, y):
    """Return sum of x and y"""
    return x + y
```

- body is indented by 4 spaces, like a `for` or `while` loop
- good practice: first line(s) are the documentation string ("docstring"), usually with triple-quotes
- if you forget what your function does, asking for help on it, e.g. `add?` in IPython prints out your docstring!
- `return` a value, or multiple values separated by comma
- arguments can be purely positional, swapping `x` and `y` in `add()` does nothing, but...

```
def subtract(x, y):
    """Return difference of x and y"""
    return x - y
```

- `subtract(x, y) != subtract(y, x)`
- can also have keyword arguments with default values:

```
def add3(x, y, z=0):
    """Add two numbers x and y, and optionally z"""
    return x + y + z
```

```
- keyword arguments ("kwargs") have to come after positional arguments
- this is invalid: `def test(x=0, y, z):`
```

- return multiple values:

```
def addsubtract(x, y):
    """Return sum and difference of x and y"""
    s = x + y
    d = x - y
    return s, d
```

- variable scope/namespaces:
    - variables defined within a function are not visible from outside the function
    - Las Vegas: what happens inside a function, stays inside a function, except for the returned result(s)
    - this is called "encapsulation", is very useful to prevent variable name clashes in your code, i.e. accidentally using the same variable name in multiple places without realizing it

⟩ coding style

- good style is easier to read, understand, debug
- try reading a book without commas, periods between sentences, or paragraphs
- official style guide: PEP 8: https://www.python.org/dev/peps/pep-0008
- main highlights from style guide:
    - variable assignment: usually leave a space on either side of an operator
        - `a = 5`, `2 + 2`, `'The year %d is here' % 2020`
        - exception: `=` in function keyword arguments should NOT have spaces on either side:
        - `def test(a, b, c=0, d=1):`
    - use only spaces for indentation, not tabs - set text editor to insert spaces on `<TAB>`
    - keep lines under 100 characters in length, 80 is preferred
        - forces you to break up excessively long lines of code into shorter pieces
        - good text editors have visual guide option that you can set at say 95 characters
    - leave a space between neighbouring function arguments, e.g. `def add(x, y):`
- comments, docstrings
    - single line: `#`
    - multiline: `"""..."""` or `'''...'''`
    - why comment? what makes a good comment?
        - mostly a message from past self to future self about what the code is, or should be, doing
        - also very nice for other people that have to read your code
        - if you change code without updating comment - confusion!
        - another form of commenting: choose descriptive variable names, use them consistently

⟩ quick function definition exercise:

1. Define a function called `product()` that takes two arguments and returns their product. Give it an appropriate docstring.

⟩ Homework 1 due next class!

⟩ extra stuff

- errors and debugging
    - `assert` allows you to quickly check assumptions that might not always hold
    - e.g. `assert a == b`
    - typical errors: `AssertionError, SyntaxError, NameError, TypeError, ValueError, IndexError, KeyError, RuntimeError, AttributeError, ZeroDivisionError`
    - set a breakpoint and "drop into debugger" with: `import pdb; pdb.set_trace()`
        - debugger commands: `l, w, s, n`

- `try, except` blocks to catch specific types of errors and deal with them
- `raise` your own errors to stop execution and inform the user of something