Announcements

- First Drop-In time:
  - Weds 1-2pm in CSIT building, Rm N113
- Week 4 – no homework this week
- Reminder – if your lab in in HN 1.25, you have a new room:
  - Weds 3-5pm: relocated to BPB W118 (tentative)
  - Fri 2-4pm: Birch 1.33 teaching lab
  - Fri 12-2pm: Birch 1.33 teaching lab

Coding Best Practices

COMP1730/6730

Have a glance at PEP8, co-authored by Guido van Rossum

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

(Extreme) example

- Working code can deliberately be made very hard to understand.
- What does this function do? Is it correct?

What is the input type?

```
def AbC(ABc):
    ABC = len(ABc)
    ABc = ABc[ABC-1:-ABC-1:-1]
    if ABC == 0:
        return 0
    abC = AbC(ABc[-ABC:ABC-1:])
    if ABc[-ABC] < 0:
        abC += ABc[len(ABc)-ABC]
    return abC
```

What is this attempted slice doing?

The function calls itself here

- Anyone?

(Extreme) example, reworked

- Now, what does this function do? Is it correct?

Helpful function name

Input name gives us a chance to guess the correct type

```
def sum_negative(input_list):
    """Return sum of all negative numbers in input_list.
    Assumes: list of numerical values. (precondition) """

    total = 0          # cumulative sum
    i = 0              # current list index
    while i < len(input_list):
        if input_list[i] < 0:
            total = total + input_list[i]
        # total now has cumulative sum of negative values
        i = i+1
    return total       # total has cumulative sum
                       # of negatives for input_list
                       # (post-condition)
```

DocString tells us what is expected in the input list

Comments form a useful narrative to what is happening

Reading other people’s code

- Or even code you wrote a while ago.
- Your primary impression will be how understandable the code is
  - PEP20 -> *Readability counts.*
  - And, do you think it does what it says it does?

- PEP8: Style Guide for Python Code (https://peps.python.org/pep-0008/)
- Python Enhancement Proposals (PEPs – https://peps.python.org):
  - Kind-of work like technical white papers on specific topics (or political position statements, sometimes. PEP20: The Zen of Python)
  - Are numbered
  - Go back twenty years sometimes (and are often written by Guido van Rossum)
  - Can be very specific. PEP257: Docstrings

Contents

- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code Lay-out
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Should a Line Break Before or After a Binary Operator?
  - Blank Lines
  - Source File Encoding
  - Imports
  - Module Level Dunder Names
- String Quotes
- Whitespace in Expressions and Statements
  - Pet Peeves
  - Other Recommendations
- When to Use Trailing Commas
- Comments
  - Block Comments
  - Inline Comments
  - Documentation Strings
- Naming Conventions
  - Overriding Principle
  - Descriptive: Naming Styles
  - Prescriptive: Naming Conventions
    - Names to Avoid
    - ASCII Compatibility
    - Package and Module Names
    - Class Names
    - Type Variable Names
    - Exception Names
    - Global Variable Names
    - Function and Variable Names
    - Function and Method Arguments
    - Method Names and Instance Variables
    - Constants

Code Lay-out

Indentation

Use 4 spaces per indentation level.

Continuation lines should align wrapped elements either vertically using Python’s implicit line joining inside parentheses, brackets and braces, or using a *hanging indent* [1]. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line:

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

The 4-space rule is optional for continuation lines.

Optional:

```
# Hanging indents *may* be indented to other than 4 spaces.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

Python Enhancement Proposals

https://peps.python.org

What is code quality?

- And, why should we care?

- Writing code is easy. Writing code do that you (and others) can be confident it is correct is not.
- You will often spend more time finding and fixing errors that you made (“bugs”) than writing code in the first place
- Good code is not only correct, but helps people (including yourself) understand what it does and why it is correct

Aspects of code quality

1. Commenting and documentation
2. Variable and function naming
3. Code organization (for large programs)

1. Comments: what makes a good comment?

- Good comments raise the level of abstraction:
  - **What** the code does and **why**, not **how**
  - Except when how is especially complex
- Describe parameters and assumptions
  - Python is dynamically typed, unlike other languages where the type must be explicitly specified:

```
def sum_negative(input_list):
    """Return sum of negative numbers in input_list.
    Assumes input_list contains only numbers."""
```

- Comments should always be up-to-date (and maintained)
- Located with relevance to their meaning

1. Comments: how NOT to comment

- Commenting is not a way to make up for poor quality in other aspects of code (organization, naming, etc):

```
x = 0 # Set the total to 0.
```

- Just plain wrong comments or not in the right place (or refers to the way your code previously did something):

```
# loop over list to compute sum
avg = sum(the_list) / len(the_list)
```

- Stating the obvious:

```
x = 5 # Sets x to 5.
```

- Or, assume that the reader is an expert python coder

1. Documentation: the function docstring

- Use these! They will appear in the interactive `help()` too
- They are the triple-quoted (`"""` or `'''`) string as the first statement inside a function definition
  - Both in modules and classes
- In the docstring, state the:
  - Purpose and limitations of the function
  - Required and optional parameters
  - Potential side effects
  - Assumptions
  - `return` value

```
def solve(f, y, lower, upper):
    """Returns x such that f(x) = y (approximately).
    Assumes f is monotone and that a solution lies in the interval
    [lower, upper] (and may recurse infinitely if not)."""
```

- It is very normal for the docstring to often be longer than all the other statements within a function code block

1. Documentation: More docstrings

- These can be freeform text, but often have a required structure in many software projects
- A guide to docstrings conventions is available as PEP257:
  https://peps.python.org/pep-0257/
- From PEP257, here is a simple example of named parameters and their description in a docstring:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
```

Some Docstrings are structured essays

- From BioPython Seq.py module:

https://github.com/biopython/biopython/blob/master/Bio/Seq.py

```
63
64  class SequenceDataAbstractBaseClass(ABC):
65      """Abstract base class for sequence content providers.
66
67      Most users will not need to use this class. It is used internally as a base
68      class for sequence content provider classes such as _UndefinedSequenceData
69      defined in this module, and _TwoBitSequenceData in Bio.SeqIO.TwoBitIO.
70      Instances of these classes can be used instead of a ``bytes`` object as the
71      data argument when creating a Seq object, and provide the sequence content
72      only when requested via ``__getitem__``. This allows lazy parsers to load
73      and parse sequence data from a file only for the requested sequence regions,
74      and _UndefinedSequenceData instances to raise an exception when undefined
75      sequence data are requested.
76
77      Future implementations of lazy parsers that similarly provide on-demand
78      parsing of sequence data should use a subclass of this abstract class and
79      implement the abstract methods ``__len__`` and ``__getitem__``:
80
81      * ``__len__`` must return the sequence length;
82      * ``__getitem__`` must return
83
84        * a ``bytes`` object for the requested region; or
85        * a new instance of the subclass for the requested region; or
86        * raise an ``UndefinedSequenceError``.
87
88        Calling ``__getitem__`` for a sequence region of size zero should always
89        return an empty ``bytes`` object.
90        Calling ``__getitem__`` for the full sequence (as in data[:]) should
91        either return a ``bytes`` object with the full sequence, or raise an
92        ``UndefinedSequenceError``.
93
94      Subclasses of SequenceDataAbstractBaseClass must call ``super().__init__()``
95      as part of their ``__init__`` method.
96      """
97
98      __slots__ = ()
99
100     def __init__(self):
101         """Check if ``__getitem__`` returns a bytes-like object."""
102         assert self[:0] == b""
103
```

Some Docstrings have code examples

- From BioPython Seq.py module:

https://github.com/biopython/biopython/blob/master/Bio/Seq.py

```
3119
3120  def complement(sequence, inplace=None):
3121      """Return the complement as a DNA sequence.
3122
3123      If given a string, returns a new string object.
3124      Given a Seq object, returns a new Seq object.
3125      Given a MutableSeq, returns a new MutableSeq object.
3126      Given a SeqRecord object, returns a new SeqRecord object.
3127
3128      >>> my_seq = "CGA"
3129      >>> complement(my_seq, inplace=False)
3130      'GCT'
3131      >>> my_seq = Seq("CGA")
3132      >>> complement(my_seq, inplace=False)
3133      Seq('GCT')
3134      >>> my_seq = MutableSeq("CGA")
3135      >>> complement(my_seq, inplace=False)
3136      MutableSeq('GCT')
3137      >>> my_seq
3138      MutableSeq('CGA')
3139
3140      Any U in the sequence is treated as a T:
3141
3142      >>> complement(Seq("CGAUT"), inplace=False)
3143      Seq('GCTAA')
3144
3145      In contrast, ``complement_rna`` returns an RNA sequence:
3146
3147      >>> complement_rna(Seq("CGAUT"))
3148      Seq('GCUAA')
```

2. **Naming:** good naming practice

- As we saw with the first extreme example, a good function name makes a huge difference to understanding what a function does
- The names of functions and variables should tell you what it does or is used for
- Variable names should not **shadow** the name of a standard type, a built-in function or a python keyword (remember Lecture 2)
- Or variables from an outer scope:

```
def a_fun_fun(int):
    a_fun_fun = 2 * int
    max = 5
    return max < int


print(a_fun_fun(1))
```

2. **Naming:** simple advice

- Your variable and function names can be long
  - Using an IDE (like Spyder) will autocomplete names
- If in doubt, use underscores. Python built-in functions and keywords rarely (never?) have underscores
- Some short names (single letters) are very familiar to experienced programmers, and are used in certain contexts:
  - Iterator indices: `i, j, k`
  - Counts: `n, m, k`
  - Coordinates: `x, y, z`
- Avoid similar (and ambigious) names in the same context
  - *eg.* `sum_of_negative_numbers` vs `sum_of_all_negative_numbers`
    - Not very clear how these are different and leads to confusion (== bugs)

3. Code Organisation

- This is fundamentally about design and abstraction
- Good code organization:
  - Avoids repetition
  - Avoids repetition
  - Avoids repetition (and uses functions)
  - Fights complexity by isolating sub-problems and encapsulation their solutions
  - Raises the level of abstraction
  - Is easy to glance through
- In python, good code organization means you use:
  - Functions
  - Modules
  - Classes

3. Code Organisation: Functions

- Functions promote abstraction
  - They separate **what** from **how**
- A good function (usually) does just **one** thing
  - And this is reflected by the function name
- Functions reduce code repetition
- Help isolate errors and bugs to a single point
- Makes code easier to maintain and change
  - Because changes happen just in one place

Wisdom from PEP8: Indentation

- Four spaces, good. Tabs, bad.

```python
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```python
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Wisdom from PEP8: Newlines and binary operators

- This may seem a little fussy, but it makes good stylistic sense:

```python
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* series: “Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations” [3].

Following the tradition from mathematics usually results in more readable code:

```python
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Wisdom from PEP8: Whitespace

- Whitespace should always be used to increase the readability of your code
  - Code that is squashed together is harder to read
  - Logical empty lines make it possible to keep related code together and distinct from other ‘thoughts’ in the code
- Use whitespace and comments together:
  - Comments can act like section headings in text
  - The code can then resemble the paragraphs, separated by whitespace

Lecture Roadmap

- Intro to Programming
- Variables
- Functions
  - The stack
  - Scope
- Flow control
  - `if`
  - `while`
  - `for`
- Strings
- Lists
- Dictionaries

Strings

COMP1730/COMP6730

Reading: Textbook chapter 8 : Alex Downey, *Think Python*, 2nd Edition (2016)

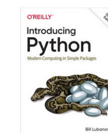
OR

Chapter 5 : Lubanovic, *Introducing Python*, 2nd Edition (2019)

But only up until section: *Search and Select*

Strings – *Think Python* Ch 8, (or *Introducing Python* - Ch 5)

- ’’’*Computer books often give the impression that programming is all about math. Actually, most programmers work with strings of text more often than numbers*’’’. Lubanovic, Ch 5
- Strings – values of type `str` in python – are used to store and process text
- A string is a **sequence** of *characters*
  - `str` is a sequence
  - Lists are another sequence



Introducing Python, 2nd Edition

Bill Lubanovic

Published by O'Reilly Media, Inc.

Python

Strings with `'`, `"` and `str()`

Australian National University

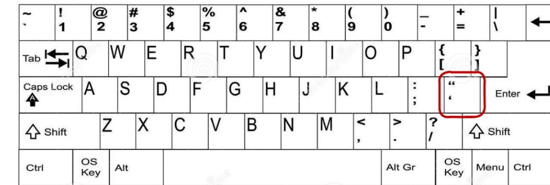
- Assign a string by placing any text between a pair of delimiters:
  - Single quotes: `tree_name = 'eucalyptus'`
  - Double quotes: `sentence = "he's going to code"`
- Explicit string creation, when it might be ambiguous:

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

Lubanovic (2019) *Introducing Python*, 2nd Ed. (Chapter 5)

The keys for quotation marks:

Australian National University



- Beware of copying-and-pasting from these slides (and PDF files or from the web).

Strings and quotation marks

Australian National University

```
>>>
>>> some_text = 'this is a string'
>>> print(some_text)
this is a string
>>> some_more_text = "this is also a string"
>>> print(some_more_text)
this is also a string
>>> some_text = 'this is a string'
>>> some_more_text = "this is a string"
>>> some_text_triple = '''this is a string'''
>>> some_text == some_more_text
True
>>> some_text == some_text_triple
True
>>> prose = '''This is
... a
... multi-line
... string
... '''
>>> print(prose)
This is
a
multi-line
string

>>>
```

'Single quotes'

"Double quotes"

'''Multi-line quotes'''

"""Another way to do it"""

Quotation marks in strings?

Australian National University

- Text often contains quotation marks too. Most programming languages have a way to get around this.
- In python, you can use the 'other' kind of quotation marks for a quick fix.

```
>>> "'Nay!' said the naysayer. 'Neigh?' said the horse."
"'Nay!' said the naysayer. 'Neigh?' said the horse."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
```

Lubanovic (2019) *Introducing Python*, 2nd Ed. (Chapter 5)

- But there is a better way… (next slide)

Escape character for quotation marks:

- You may use the backslash character '\' to **escape** your quotation marks:

```
>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

Lubanovic (2019) *Introducing Python*, 2nd Ed. (Chapter 5)

- This is a way of being explicit that the next character after the backslash should be interpreted in a certain way.
- For \" and \' escape characters, this means that the quotation should be interpreted literally as a ' or ". Not as a string delimiter.

More Escape characters

- When you need to be explicit that a character should be included in a string literally, you can use the escape character '\'
- Common escape characters (there are many more too, try \b):

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Table 6.1 - Sweigart (2019) *Automate the boring stuff with python*

```
>>>
>>> print('one\ntwo')
one
two
>>> print('one\ttwo')
one     two
>>>
```

Escape characters for newlines:

To put a newline (carriage return) into a string, use '\n':

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

Lubanovic (2019) Introducing Python, 2nd Ed.

Combining strings and string interpolation

- Strings can be concatenated with the + operator:

```
>>> name = 'Al'
>>> age = 4000
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
```

Sweigart (2019) *Automate the boring stuff with python (Chapter 6)*

- There is another short-hand syntax to do this that you may see, called **string interpolation**:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' % (name, age)
'My name is Al. I am 4000 years old.'
```

Sweigart (2019) *Automate the boring stuff with python (Chapter 6)*

Strings are sequences (reminder)

- Each of the characters in a string may be treated individually. Because `str` variables are **sequences**.
- To access each character in a string, you use the index value (enclosed in square brackets []:

```
>>>
>>> some_text = "Hello, world!"
>>> some_text
'Hello, world!'
>>> some_text[0]
'H'
>>> some_text[5]
','
>>> some_text[7]
'w'
>>>
```

- Index values always start counting from zero!

Strings are **immutable**

- Once a string is assigned, it can only be changed by re-assigning the whole string.
- If we try to change an element, we get an error:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

- If we want to change this character, we need to reassign the string:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Downey (2015) *Think Python*, 2nd Ed. (Chapter 8)

Strings and the `in` operator

- The keyword `in` can be used as a Boolean operator to test if a substring appears in another word:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

Downey (2015) *Think Python*, 2nd Ed. (Chapter 8)

`in` with `for` - string traversal

- The `in` keyword can also be used with `for` to iterate through a string:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

Downey (2015) *Think Python*, 2nd Ed. (Chapter 8)

- Output:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Example: `in`, `for` and string traversal

- And this is useful, for example – define a function to find common letters in words:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

```
>>> in_both('apples', 'oranges')
a
e
s
```

Downey (2015) *Think Python*, 2nd Ed. (Chapter 8)

Exercises

- Exercises 8-1, 8-2 and 8-4, *Think Python* Ch. 8
- Exercises in Lutz Ch 5 are a little different to what we've seen

Reading

- Lutz (2019) *Introducing Python*, Ch 5 (until section: *Search and Select*) OR
- *Think Python* Ch 8