lec2notes

February 14, 2022

0.1 Review from last week + homework

- parentheses and order of operations
- more than one way to skin a cat, but one way is usually best
- waltlabtools and Mito install issues
  - Your computer is literally haunted
  - Username is a resolvable URL
  - Transitions Lenses go dark
-

[ ]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

0.2 `str`

- single or double quotes
- concatenate strings with `+` or nothing
- `len` for length of string

- `str(var)` to convert `var` to a string
- indexing strings with `[]`
  - indexing for a specific letter
  - slicing
- `type()`

[ ]:
```
import numpy as np

str1 = "hello world"
len(str1)
str2 = "hello" + "world"
str2 = "hello" "world"
len(str2)

str3 = "abcdefg"
len(str3)
str3[2]
str3[2:5]
str3[2:]
str3[:5]
str3[-1]
str3[2:-1]
print("pi is approximately " + np.pi) # raises error
print("pi is approximately " + np.pi)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/tdougan/Dropbox (HMS)/Research/General/Scientific Computing/Python Class
 ↪pythonminicourse/lec2notes.ipynb Cell 2' in <module>
      <a href='vscode-notebook-cell:/Users/tdougan/Dropbox%20%28HMS%29/Research/
 ↪General/Scientific%20Computing/Python%20Class/pythonminicourse/lec2notes.
 ↪ipynb#ch0000004?line=14'>15</a> str3[-1]
      <a href='vscode-notebook-cell:/Users/tdougan/Dropbox%20%28HMS%29/Research/
 ↪General/Scientific%20Computing/Python%20Class/pythonminicourse/lec2notes.
 ↪ipynb#ch0000004?line=15'>16</a> str3[2:-1]
---> <a href='vscode-notebook-cell:/Users/tdougan/Dropbox%20%28HMS%29/Research/
 ↪General/Scientific%20Computing/Python%20Class/pythonminicourse/lec2notes.
 ↪ipynb#ch0000004?line=16'>17</a> print("pi is approximately " + np.pi) # raise␣
 ↪error
      <a href='vscode-notebook-cell:/Users/tdougan/Dropbox%20%28HMS%29/Research/
 ↪General/Scientific%20Computing/Python%20Class/pythonminicourse/lec2notes.
 ↪ipynb#ch0000004?line=17'>18</a> print("pi is approximately " + np.pi)

TypeError: can only concatenate str (not "float") to str
```

0.3 `list`

- create a list with `list()` or `[]`

- lists can be indexed and sliced
- lists are mutable
- concatenate lists with `+` or `list.extend()`
  - `list.extend()` vs `list.append()`
- lists can contain anything, even other lists
- so when we wrote `np.array([...])`, we were saying to make a list of numbers, and then turn it into an array

[ ]:
```
list1 = []  # empty list
list2 = [1, 5, np.pi, "f"]
list3 = list(str3)

list3[2] = "C"
list3.append("h")
list3.extend(["i", "j", "l"])
list3.append(["i", "j", "k"])
```

0.4 `tuple`

- tuples are immutable but their contents don't have to be
- create tuples by separating arguments by commas
- unpacking

[ ]:
```
tuple1 = ()
tuple2 = 1, "f", 7
tuple3 = tuple(list2)
x, y, z = "x", "y", "z"
```

[ ]:
```
tuple
```

0.5 `dict`

- dictionaries have keys and values
  - keys have to be unique and immutable

[ ]:
```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
tel
tel['jack']
del tel['sape']
tel['irv'] = 4127

tel.keys()
tel.values()
```

0.6 `if` statements

- basic structure and syntax: `if`, `else`, `elif`
- examples of tests

- comparisons
  - `is` vs `==`
- automatically convert to boolean
  - `bool()`
  - `and`, `or`, `not`
  - numbers 0
  - non-empty collections
- `isinstance()`
- `in`

[ ]:
```
if 5 < 7:
    print("math works")

if 5 < 7:
    print("math works")
else:
    print("math does not work")

if 5 < 7:
    print("math works")
elif 5 == 7:
    print("math is broken in a very specific way")
else:
    print("math does not work")

print(5 < 7)

bool(0)
bool([])
bool(7)
bool("hello world")

x = True

if x == True:
    print(x)

if x is True:
    print(x)

if x:
    print(x)

# ask the group: how to test if y is a list?
if type(y) == list:  # bad
    pass # noqa
if type(y) is list:  # slightly better?
```

```
    pass # noqa
if isinstance(y, list):  # good
    pass # noqa

1 in [1, 2, 3]
"l" in "hello world"
"lo" in "hello world"
```

0.7 `for` loops

- `range()`
- looping over tuples and lists
- looping over dictionaries: `dict.items()`
- `enumerate()`
- don't modify what you're looping over unless you want your life to suck

[ ]:
```
for i in range(10):
    print(i)

for i in range(4, 10):
    print(i)

for i in range(4, 10, 2):
    print(i)

for i in [4, 6, 8]:
    print(i)


for n, l in enumerate("abcdefg"):
    print(n, l)
# what does this code do?


for key, value in tel.items():
    print(value)

for key in tel.keys():  # or for key in tel:
    print(tel[key])

for value in tel.values():
    print(value)
```

0.8 functions

[ ]:
```
def identity_function(x):
    """Returns its argument, unchanged."""
    return x

def print_type(x):
    """Prints the type of its argument."""
    x_type = type(x)
    print(x_type)

def is_equal(x, y):
    """Returns True if x==y; False otherwise."""
    x_equal_y = x == y
    return x_equal_y

def is_equal(x, y):
    """Returns True if x==y; False otherwise."""
    return x == y


def fon(aeb_):
    """The fraction of beads which are on.
    Converts the average enzymes per bead (AEB) to the fraction of
    on-beads (fon) using Poisson statistics. The formula used is
    `fon_` = 1 - exp(-`aeb_`).

    Parameters
    ----------
    aeb_ : numeric or array-like
        A scalar or array of the average number of enzymes per bead.

    Returns
    -------
    fon_ : same as input, or array
        The fractions of beads which are "on."

    See Also
    --------
    aeb : inverse of fon

    """
        return 1 - np.exp(-aeb_)

def limit_of_detection(blank_signal, inverse_fun=None, sds=3):
    """Computes the limit of detection (LOD).

    Parameters
```

```
    ----------
    blank_signal : array-like
        Signal (e.g., average number of enzymes per bead, AEB) of the
        zero calibrator. Must have at least two elements.
    inverse_fun : ``function`` or ``CalCurve``
        The functional form used for the calibration curve. If a
        function, it should accept the measurement reading (`y`, e.g.,
        fluorescence) as its only argument and return the value (`x`,
        e.g., concentration). If **inverse_fun** is a ``CalCurve``
        object, the LOD will be calculated from its ``inverse`` method.
    sds : numeric, optional
        How many standard deviations above the mean should the
        background should the limit of detection be calculated at?
        Common values include 2.5 (Quanterix), 3 (Walt Lab), and 10
        (lower limit of quantification, LLOQ).

    Returns
    -------
    lod_x : numeric
        The limit of detection, in units of x (e.g., concentration).

    """
    mean = np.mean(blank_signal)
    stdev = np.std(blank_signal)
    lod_y = mean + sds * stdev
    lod_x = inverse_fun(lod_y)
    return lod_x
```

0.9 lambda

[ ]:
```
fon = lambda aeb_: 1 - np.exp(-aeb_)
```

0.10 comprehensions

- list comprehensions
- dictionary comprehensions
- comprehensions are good if they fit on one line; otherwise, use a for loop

[ ]:
```
nums = [1, 2, 3, 4, 5, 6]
# how would I make a list of each of these numbers squared?

nums_squared = []
for n in nums:
    nums_squared.append(n**2)

nums_squared = [n**2 for n in nums]
```

```
dict_of_nums_squared = {n: n**2 for n in nums}
```

0.11 Brief demo: how waltlabtools is supposed to work

[ ]:
```
import waltlabtools as wlt
import pandas as pd

run_history = wlt.read_raw_hdx()
# navigate to ~/Dropbox (HMS)/Research/General/Presentations/2022-01 HIV Update/
 ↪run_history.csv
gag_points = run_history[(run_history.Plex == "td Gag p24 singleplex beads") &␣
 ↪(run_history["Replicate Conc."] < 1e3)][["Replicate Conc.", "Replicate AEB"]]
gag_cc = wlt.CalCurve.from_data(x=gag_points["Replicate Conc."],␣
 ↪y=gag_points["Replicate AEB"], model="4PL")
gag_cc.plot(xlabel="Gag p24 (pg/mL)", ylabel="AEB")
print(gag_cc.lod)
```