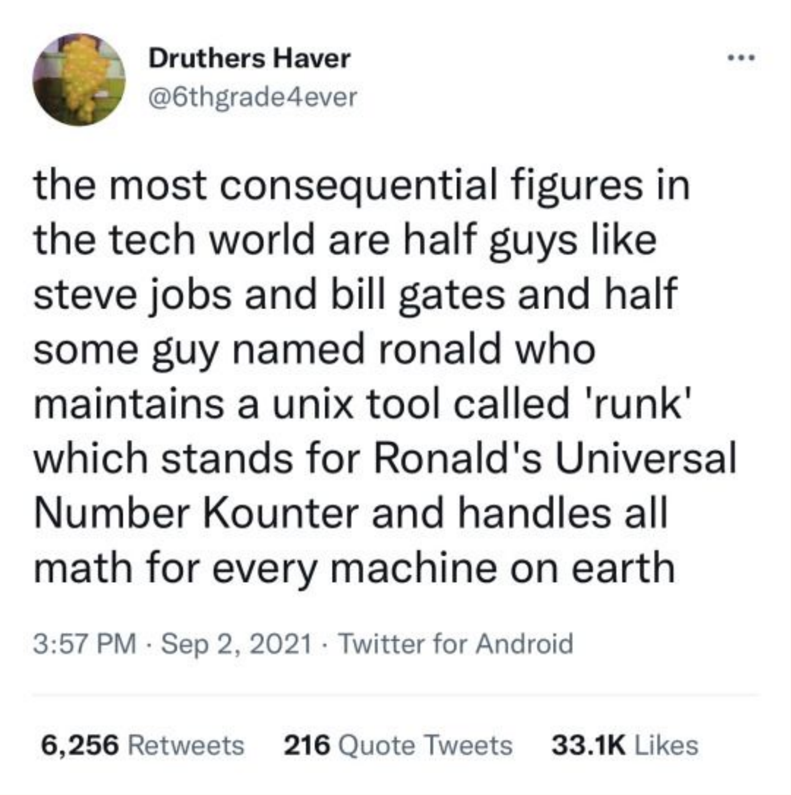
LING 331:
Text Processing for Linguists

Week 3

Basic Python 1

Abstraction is wonderful! ... and terrifying.



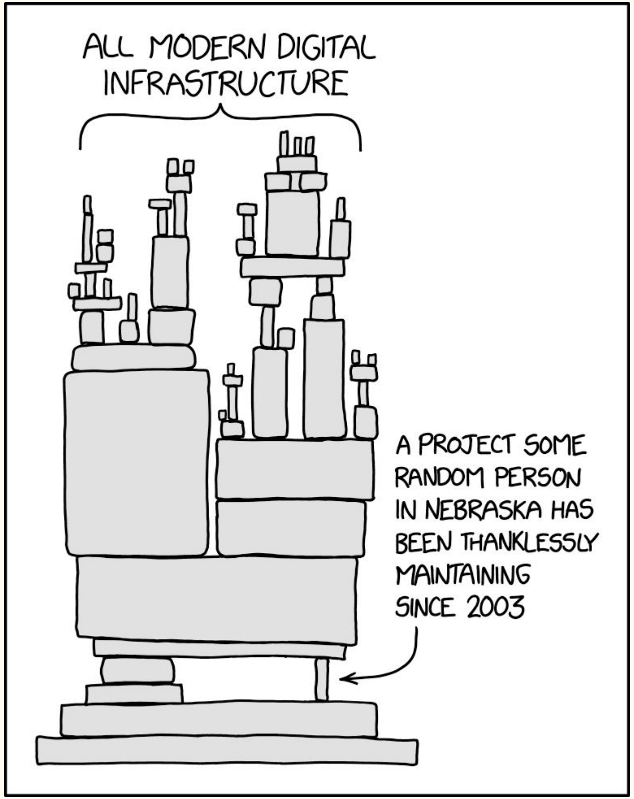
Druthers Haver
@6thgrade4ever

the most consequential figures in the tech world are half guys like steve jobs and bill gates and half some guy named ronald who maintains a unix tool called 'runk' which stands for Ronald's Universal Number Kounter and handles all math for every machine on earth

3:57 PM · Sep 2, 2021 · Twitter for Android

6,256 Retweets 216 Quote Tweets 33.1K Likes

xkcd.com


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

Notes from Assignment 2

- Regular expressions are tricky --
  these are the pattern inputs to sed and grep --
  we will go over in much more detail later in the quarter!

Notes from Assignment 2

- Whitespace is invisible and therefore tricky
  e.g. top word = 46401 instances of ‘ ’
  Can run another sed to remove this, or a one-command fix:

  ```
  sed 's/ +/\n/g'
  ```

- Similar, `sed '/^$/d'` works but misses lines with spaces

- `[0-9]` is all digits (doesn’t work to do e.g. `[0-100]`)

Notes from Assignment 2

Quoting!

- Be very careful with quoting! And (), [], etc.
  Each ' requires another ' to close it,
  each " requires another " to close it.

- Syntax highlighting helps a lot.

Notes from Assignment 2

Quoting!

- Double quotes interpret arguments (e.g. `"$1"`) and escapes, Single quotes leave them be. https://stackoverflow.com/questions/6697753/difference-between-single-and-double-quotes-in-bash
- Whitespace (spaces, tabs, newlines) is interpreted as a delimiter between arguments! (See TLCL Ch. 7)

Notes from Assignment 2

Stream Management!

- Be aware that almost all text filter commands can accept the input file as an argument (e.g. `sed 's/sad/happy/g' input.txt`)
- Careful with > (write) vs. >> (append)
- > and >> end the stream (alternatively can use `tee`)

Notes from Assignment 2

- Better to not generate auxiliary files, e.g.:

```
    grep love shakes.txt > lovelines.txt
    wc -l lovelines.txt
```

- This works, but adds cruft and obscures things later - if we come back in a day, how exactly did we get `lovelines.txt`? Once it's created we lose the "story," if you will.
  Thus piping!

```
    grep love shakes.txt | wc -l
```

Notes from Assignment 2

- Don’t call programs like nano / less from a script:
  it’ll stop execution of the script until you close that instance.
  nano/less are not text filters like grep/sed/tr/sort/etc.
  - They can *receive* input from stdin,
    they just don’t pass it through to stdout

- This and all further assignments should be runnable!
  (don’t write the answer, write the code that generates it)

Notes from Assignment 2

- If you tried “Unix for Poets” you may have encountered some version differences!

- The standard `tr` worked differently some years ago!

- Welcome to version differences - an eternal problem.

… and now for something completely different!

Welcome to Python world!

What is the “stuff” of programming?

Generally, we are **manipulating data** in ever-more-complex ways

We think of that data as a set of objects, like objects in the real world

Variable Names are symbolic names that point to persistent bits of data

(a lot like file names)

Variable Types define different sorts of data

Numeric

integer

`42`

float

`42.0`

Sequence

list

`['y', 2, False]`

tuple

`(6, ‘b’, 19.7)`

Text

string

`'hello!'`

None

`None`

Truthy

boolean

`True, False`

(next week)

Set set

Mapping dict{}

Statements are units of code that do something

Assignment (=)

```
year = 2020 # integer


mssg = 'hooray!' # string


e = 2.71828 # float
```

Statements are units of code that do something

Equality Testing `(==, !=, >, <, >=, <=)`

```
>>> year != 2016
True

>>> mssg == 'howdy!'
False

>>> e <= 3
True
```

Statements are units of code that do something

Arithmetic (+, -, *, /, **)

```
>>> year * 3
6060

>>> 'hip hip ' + mssg
'hip hip hooray!'

>>> e / 2
1.35914
```

Statements are units of code that do something

Incrementing (arithmetic plus assignment)

```
>>> year += 18
>>> year
2038

>>> mssg *= 5
>>> mssg
'hooray!hooray!hooray!hooray!hooray!'
```

Functions - a three-step process

1. *Take some input*

   Often called “arguments” to the function (can be no args)

2. *Do some computation*

   Often called the “body” of the function

3. *Produce some output*

   Often called “return”ing data (can be `None`)

Functions take input, do some computation, produce output

Important Built-ins 1

```
print(x) # print representation of x

help(x) # detailed help on x

type(x) # return type of x

dir(x) # list methods and attributes of x
 (methods are functions bound to objects)
 (attributes are variables bound to objects)
```

Functions take input, do some computation, produce output

Important Built-ins 2

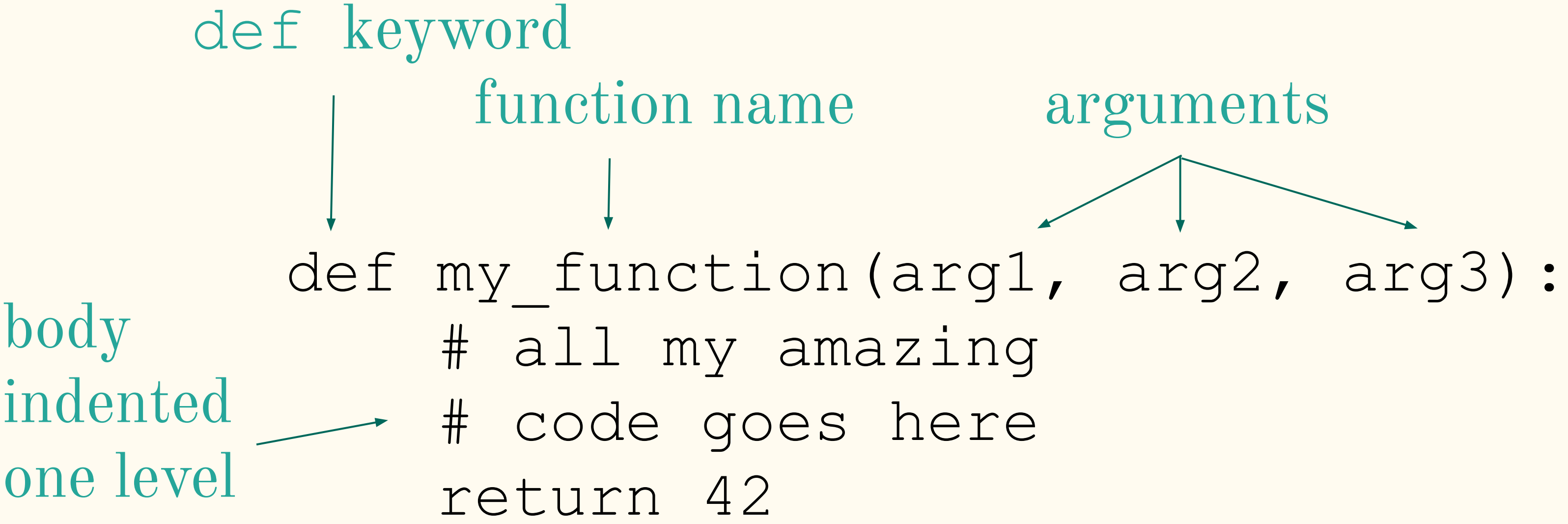
```
sorted(x) # return sorted version of x

min(x), max(x) # mathematical operations
sum(x)         # on sequences

int(x), float(x), bool(x) # 'casting', a.k.a.
list(x), tuple(x), str(x) # type conversion
```

Functions take input, do some computation, produce output

Defining New Functions



`def` keyword

function name

arguments

body indented one level

```
def my_function(arg1, arg2, arg3):
    # all my amazing
    # code goes here
    return 42
```

Control Flow organizes the order code executes

Conditionals - **if**, **elif**, **else** - enter section if condition is met

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     print('Negative!')
... elif x == 0:
...     print('Zero!')
... else:
...     print('Positive!')
Positive!
```

Control Flow organizes the order code executes

Loops - **for ... in** - loop over items of a sequence

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Control Flow organizes the order code executes

Loops - **for ... in** - loop over numbers by using **range**

```
>>> for i in range(5):
...     print(i)
…
0
1
2
3
4
```

Control Flow organizes the order code executes

Loops - **for ... in** - for reading lines in a file with **open**

```
>>> for line in open('shakes.txt'):
...     print(line)
1609

THE SONNETS

by William Shakespeare
```

Control Flow organizes the order code executes

Loops - **while** - loop until condition is met

```
>>> # Fibonacci: sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a, end=' ')
...     a, b = b, a+b
... print('')
...
0 1 1 2 3 5 8
```

Whitespace is obligatory for demarcating code blocks

The body of

function definitions

and

control flow elements

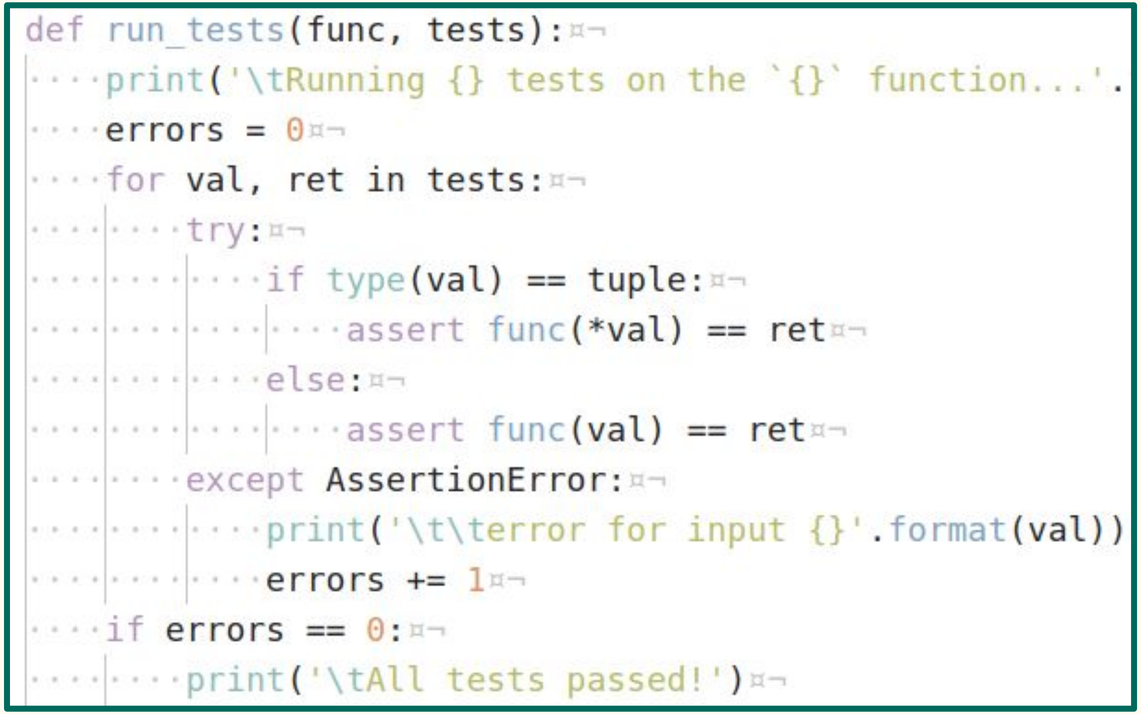
must be indented

by one level

Recommended to be

--\t-- one tab

. . . . or four spaces



```
def run_tests(func, tests):
    print('\tRunning {} tests on the `{}` function...'.
    errors = 0
    for val, ret in tests:
        try:
            if type(val) == tuple:
                assert func(*val) == ret
            else:
                assert func(val) == ret
        except AssertionError:
            print('\t\terror for input {}'.format(val))
            errors += 1
    if errors == 0:
        print('\tAll tests passed!')
```

Whitespace is obligatory for demarcating code blocks

- Most text editors deal with whitespace semi-intelligently

- E.g., emacs sees that a file ends in .py,
  and interprets the text as python code (syntax highlighting)
  and tries to make the whitespace consistent

- Pressing the [Tab] key will jump to the logical indent.
  But be careful e.g. closing control flow statements,
  try pressing [Tab] multiple times.

String and List Indexing

```
>>> job_title = 'LINGUIST'
```

Char (or List Item)	L	I	N	G	U	I	S	T
Index	0	1	2	3	4	5	6	7
Reverse Index	-8	-7	-6	-5	-4	-3	-2	-1

Syntax:
sequence[start:end]

```
>>> job_title[3:-1]
'GUIS'    # inclusive of start, not inclusive of end

>>> job_title[:5]
'LINGU'    # can leave off start or end
```

Object-oriented Programming

We categorize real-world objects by their

properties (facts about them)
“Scissors have two loops to hold
and two blades that open
when you separate the loops.”

and **affordances** (what we can do with them)
“We use scissors to cut things.”

Object-oriented Programming

In Python, objects of a certain type have certain **attributes** (associated variables/metadata) and **methods** (associated functions)

```
>>> lil_snippy = PairOfScissors()
>>> lil_snippy.size
15
>>> lil_snippy.cut(robs_finger)
“Ow!”
```

Object-oriented Programming

In Python (and many other OOP languages), everything is officially an object. Even functions!

Many types come with very informative attributes and useful methods!

OOP is a “programming paradigm.” There are others! At this stage you don’t need to worry about that.

String Methods are functions associated with string objects

strip, rstrip, lstrip

```
>>> s = '  my sTrInGggg!\n'
>>> s = s.strip()
>>> s
'my sTrInGggg!'
>>> s = s.strip('!').strip('g')
>>> s
'my sTrInG'
```

upper, lower

```
>>> s = s.lower()
>>> s
'my string'
```

find

```
>>> s.find('str')
3
```

replace

```
>>> s.replace('my','your')
'your string'
```

startswith, endswith

```
>>> s.startswith('balloon')
False
```

List Methods are functions associated with list objects

append

```
>>> x = [1, 4, 9, 16]
>>> x.append(9)
>>> x
[1, 4, 9, 16, 9]
```

index

```
>>> x.index(4)
1
```

remove deletes the first occurrence

```
>>> x.remove(9)
>>> x
[1, 4, 16, 9]
```

pop removes and returns the last element

```
>>> x.pop()
9
>>> x
[1, 4, 16]
```

Strings and Lists

Strings are like sequences of characters

Key difference: lists are mutable — can be changed

```
my_list[3] = 'yes'
```
☺

strings are immutable — cannot be changed

```
my_str[3] = 'n'
```
✖

String methods to convert to/from lists

split

```
>>> s = 'my string'
>>> s.split()
['my', 'string']
```

join

```
>>> ' '.join(['your','string'])
'your string'
```

Assignment Walkthrough

Answers are short but can be tricky!

Think *Decomposition*

how can I break this into smaller, doable sub-problems?

Tests provided after each function! (non-exhaustive)

Assignment Walkthrough

You **must** do:

```
   module load python/anaconda3.6
```

every time you login to Quest

(or include this line in your .bashrc)

Run the assignment with:

```
  python assignment3.py
```

The assignment **must** run when you are done!