Lesson 4: Type Conversion, Mutability, Sequence Indexing

Fundamentals of Text Processing for Linguists

Na-Rae Han

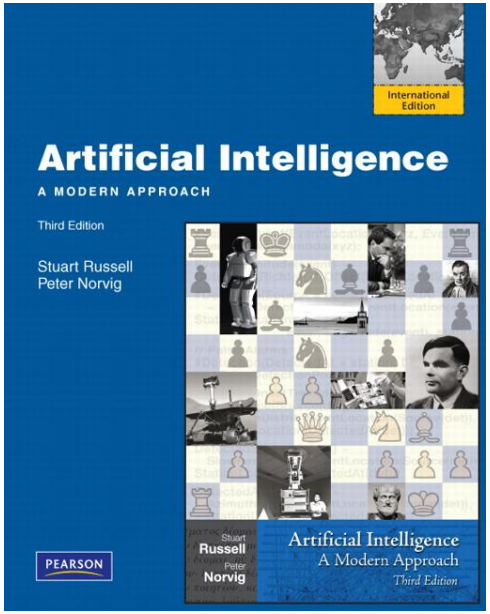
Objectives

- Python data types
- Mutable vs. immutable object types
  - How variable assignment and reference works
- Type conversion
  - Conversion between types
- Sequence types: string, list, tuple
  - Indexing
  - Slicing
  - in, +, * operations

* Today's slides borrowed heartily from http://tdc-www.harvard.edu/Python.pdf

All about palindromes

- Fun reads:
  - "Doubling in the Middle", Sep 2011, *The Believer*
    http://www.believermag.com/issues//201109/?read=article_kornbluh
  - "World's Longest Palindrome Sentence?" by Peter Norvig
    http://norvig.com/palindrome.html
    - He uses Python scripts to generate palindromes using word lists – his record is 17,826 words!



List is MUTABLE, string is not

- `.reverse()`

```
>>> mwords
['Mary', 'had', 'a', 'little', 'lamb']
>>> mwords.reverse()
>>> mwords
['lamb', 'little', 'a', 'had', 'Mary']
>>> mwords.reverse()
>>> mwords
['Mary', 'had', 'a', 'little', 'lamb']
```

Reverses a list **IN PLACE**: **original list is changed**

List is **MUTABLE**
String is **IMMUTABLE**

```
>>> wd = 'school'
>>> wd.upper()
'SCHOOL'
>>> wd
'school'
```

cf. String operations **do not change the original string**!

Python data types

```
33
```

`int`: integer

```
5.49
```

`float`: floating point number

```
'Bart'
'Hello, world!'
```

`str`: string (a piece of text)

```
['cat', 'dog', 'fox', 'hippo']
```

`list`

```
('Spring', 'Summer', 'Winter', 'Fall')
```

`tuple`

```
{'Homer':36, 'Marge':36, 'Bart':10, 'Lisa':8, 'Maggie':1}
```

`dict`: (dictionary) maps a value to an object

Immutable data types

int	`33`
float	`33.5`
str	`'Hello, world!'`
tuple	`('Spring', 'Summer', 'Winter', 'Fall')`

- In Python, the data types **integer**, **float**, **string**, and **tuple** are ***immutable***.
  - Python functions do NOT directly change these data – they are unchangeable!
    - Instead, a new object is created in memory and returned.
  - Value change only occurs through a <u>new assignment statement</u>.

```
>>> x = 'hello'
>>> x.upper()
'HELLO'
>>> x
'hello'
>>> x = x.upper()
>>> x
'HELLO'
```

Mutable data types

list `['cat', 'dog', 'fox', 'hippo']`

dict `{'Homer':36, 'Marge':36, 'Bart':10, 'Lisa':8}`

- **List** and **dictionary** are ***mutable*** data types.
  - When a Python method* is *called on* these data, the operation is done **in place**.
    ← The original data in memory are changed!
  - They are *not* copied into a new memory address each time.

```
>>> li = [13,5,4,2]
>>> li.sort()
>>> li
[2, 4, 5, 13]
```

* "**Methods**" are functions that are specific to a data type. They are "**called on**" a data object and have `object.method()` syntax.

Assignment: under the hood

- Binding a variable in Python means setting a name to hold a reference to some object.
  - Assignment creates *references*, not copies.
- Names in Python do not have an intrinsic type; objects do.
  - Python determines the type of the reference automatically based on the data object assigned to it.
- You create a name the first time it appears on the left side of an assignment statement:
  - x = 3
- A reference is deleted via garbage collection after any names bound to it have passed out of scope.

Understanding reference semantics

- Assignment manipulates *references*
  - var1 = var2 **does not make a copy** of the object var2 references
  - var1 = var2 makes var1 **reference** the object var2 references!

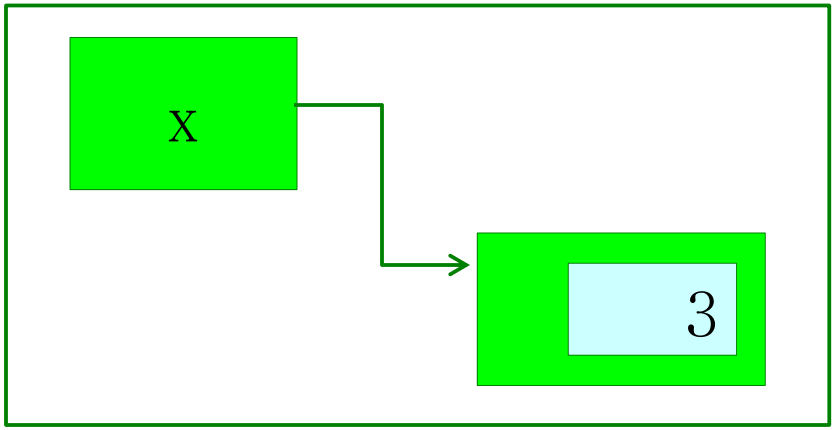
- So, for immutable data types (integers, floats, strings) assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3
>>> x = 4          # Creates ref for 4. Change x to point to it
>>> print y        # No effect on y, which still points to 3
3
```

Reference vs. immutable types

- So, for immutable datatypes (integers, floats, strings) assignment behaves as you would expect:

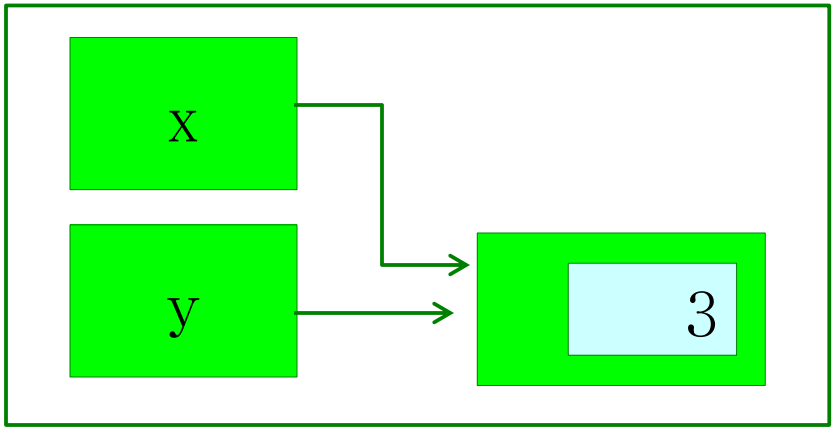
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3
>>> x = 4          # Creates ref for 4. Change x to point to it
>>> print y        # No effect on y, which still points to 3
3
```



Reference vs. immutable types

- So, for immutable datatypes (integers, floats, strings) assignment behaves as you would expect:

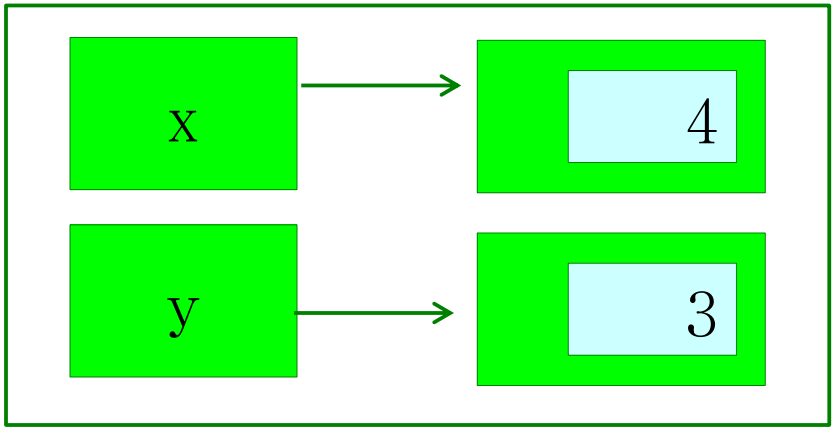
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3
>>> x = 4          # Creates ref for 4. Change x to point to it
>>> print y        # No effect on y, which still points to 3
3
```



Reference vs. immutable types

- So, for immutable datatypes (integers, floats, strings) assignment behaves as you would expect:

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3
>>> x = 4          # Creates ref for 4. Change x to point to it
>>> print y        # No effect on y, which still points to 3
3
```



Reference vs. mutable types

- **Mutable** data types (list, dictionaries) behave differently!
  - Method functions change these data **in place**, so…

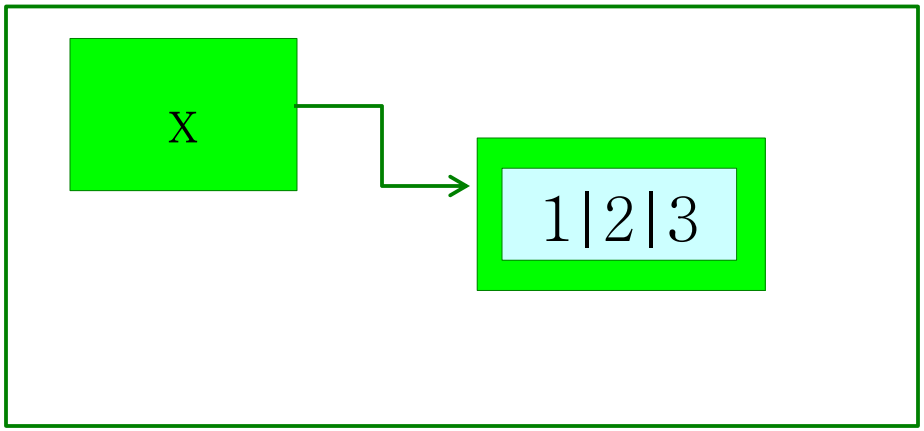
```
>>> x = [1,2,3]          # x references list [1,2,3]
>>> y = x                # y now references what x references
>>> x.append(4)          # this changes the original list in memory
>>> print x
[1, 2, 3, 4]
>>> print y
[1, 2, 3, 4]
```

Value of *y* also changed!

Reference vs. mutable types

- **Mutable** data types (list, dictionaries) behave differently!
  - Method functions change these data **in place**, so…

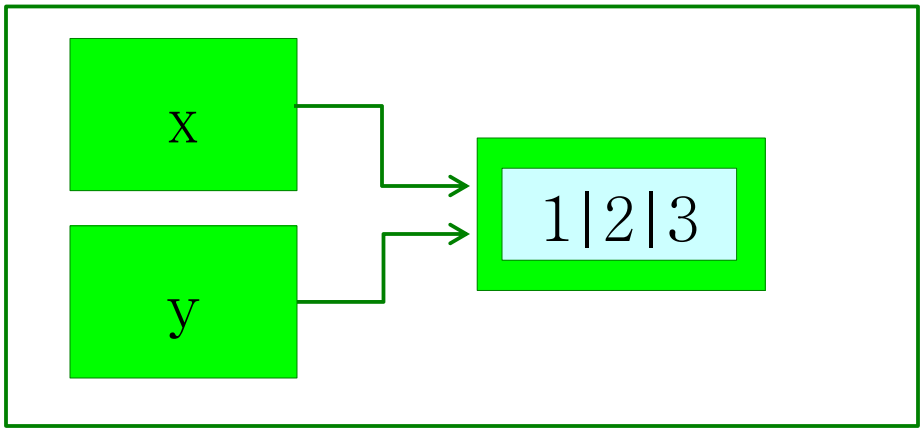
```
>>> x = [1,2,3]          # x references list [1,2,3]
>>> y = x                # y now references what x references
>>> x.append(4)          # this changes the original list in memory
>>> print x
[1, 2, 3, 4]
>>> print y
[1, 2, 3, 4]
```



Reference vs. mutable types

- **Mutable** data types (list, dictionaries) behave differently!
  - Method functions change these data **in place**, so…

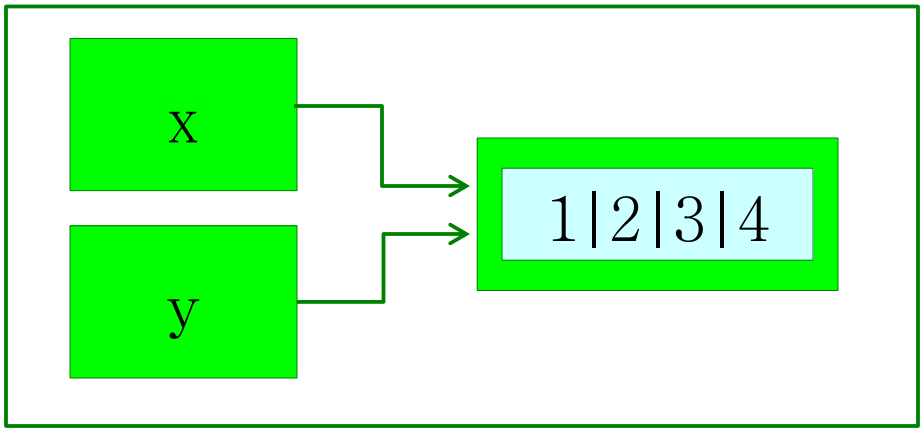
```
>>> x = [1,2,3]          # x references list [1,2,3]
>>> y = x                # y now references what x references
>>> x.append(4)          # this changes the original list in memory
>>> print x
[1, 2, 3, 4]
>>> print y
[1, 2, 3, 4]
```



Reference vs. mutable types

- **Mutable** data types (list, dictionaries) behave differently!
  - Method functions change these data **in place**, so…

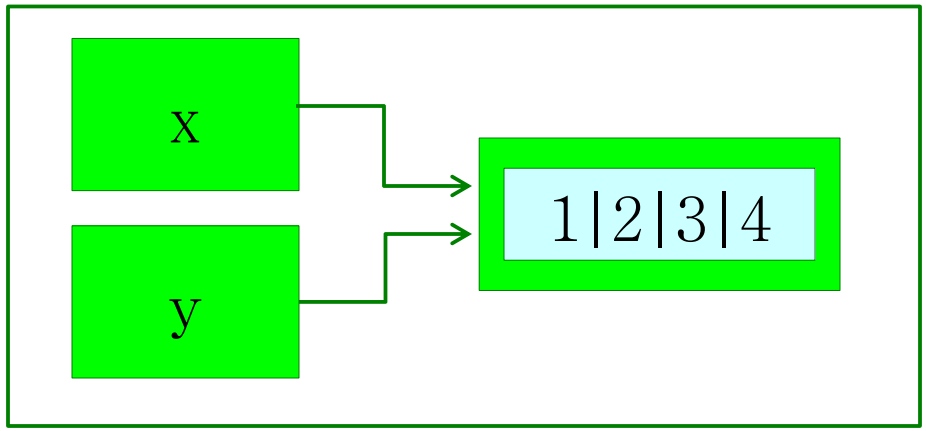
```
>>> x = [1,2,3]          # x references list [1,2,3]
>>> y = x                # y now references what x references
>>> x.append(4)          # this changes the original list in memory
>>> print x
[1, 2, 3, 4]
>>> print y
[1, 2, 3, 4]
```



Reference vs. mutable types

- **Mutable** data types (list, dictionaries) behave differently!
  - Method functions change these data **in place**, so…

```
>>> x = [1,2,3]          # x references list [1,2,3]
>>> y = x                # y now references what x references
>>> x.append(4)          # this changes the original list in memory
>>> print x
[1, 2, 3, 4]
>>> print y
[1, 2, 3, 4]
```



x and *y* refer to **the same object in memory**! When *x* is modified, *y* also changes

== vs. is

```
>>> str1 = 'cat'
>>> str2 = 'cat'
>>> str1 == str2
True
>>> str1 is str2
True

>>> list1 = ['cat', 'dog']
>>> list2 = ['cat', 'dog']
>>> list1 == list2
True
>>> list1 is list2
False
>>> list1 = list2
>>> list1 is list2
True
```

==
tests for equivalence of **value**

is
tests for equivalence of the **object in memory**

Data types in Python

`type()` displays the data type of the object

```
>>> h = 'hello' # h is str type
>>> h = list(h) # h now refers to a list
>>> h
['h', 'e', 'l', 'l', 'o']
>>> type(h)
<type 'list'>
```

- Beware of **type conflicts**!

```
>>> w = 'Mary'
>>> w+3
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    w+3
TypeError: cannot concatenate 'str' and 'int' objects
```

Converting between types

`int()` string, floating point → integer

```
>>> int(3.141592)
3
```

`float()` string, integer → floating point number

```
>>> float('256')
256.0
```

`str()` integer, float , list, tuple, dictionary → string

```
>>> str(3.141592)
'3.141592'
>>> str([1,2,3,4])
'[1, 2, 3, 4]'
```

`list()` string, tuple, dictionary → list

```
>>> list('Mary')
['M', 'a', 'r', 'y']
```

Practice: Fix this script!

2 minutes

- "Tip calculator" script:

```
b = raw_input('What\'s your bill amount? ') # b is str

tip = b * 0.15
total = b * 1.15
```

- Open your **IDLE <u>editor</u>**, and try it out.
  - The script is broken… fix it!

Fixed: type conversion is key

tip_calc2.py - F:/tip_calc2.py

File Edit Format Run Options Windows Help

```
#-----------------------------------------------------------
# tip_calc2.py
# Demonstrates type conversion, successfully this time
#-----------------------------------------------------------

b = raw_input('What\'s your bill amount? ') # b is str
b = float(b)                               # b is now a float

tip = b * 0.15
total = b * 1.15

summary = 'Your tip amount is $' + str(tip) + '\n' \
          + 'Your total payment is $' + str(total)
             # amounts are now in str type

print summary
```

Ln: 19 Col: 0

Don't forget **Space** '␣' at the end of the prompt string!

Sequence types

- Three Python basic data types are based on **sequence**:
  - **Strings**, **lists**, and **tuples**

str	`'Hello, world!'`
list	`['cat', 'dog', 'fox', 'hippo']`
tuple	`('Spring', 'Summer', 'Winter', 'Fall')`

  - These sequence types share much of the same syntax and functionality.
  - The operations shown in this section can be applied to *all* sequence types

Sequence types

1. **String**

```
'hello'
```

   - A sequence of individual characters
   - ***Immutable***

2. **List**

```
[1, 2, 'ab', 3.14]
```

   - An ordered sequence of items
   - Items can be mixed types.
   - ***Mutable***: items can be changed, added or removed

3. **Tuple**

```
(1, 2,'ab', 3.14)
```

   - A fixed sequence of mixed types of items
   - Sort of like a list, but *cannot be altered*
   - ***Immutable***!

Indexing

- Individual items of a string, list, or tuple can be accessed using square bracket "array" notation: [*index*]
- Index starts from **0**!

```
>>> st = 'Hello, world!'
>>> st[1]
'e'
>>> li = [1, 2, 'ab', 3.14]
>>> li[1]
2
>>> tu = ('Homer', 'Marge', 'Bart', 'Lisa')
>>> tu[1]
'Marge'
```

Positive vs. negative indices

```
>>> st = 'Hello!'
>>> li = ['Mary', 'had', 'a', 'little', 'lamb']
```

- **Positive index**: count from the left, starting with 0

```
>>> st[1]
'e'
>>> li[1]
'had'
```

- **Negative index**: count from right, starting with -1

```
>>> st[-1]
'!'
>>> li[-1]
'lamb'
```

H	e	l	l	o	!
0	**1**	2	3	4	5
-6	-5	-4	-3	-2	**-1**

Slicing returns a copy of a subset

- Slicing syntax: `s[start:end]`
  - Returns the elements beginning at *start* and extending up to but **not including** *end*
  - ← Copying starts at the first index, and stops copying **before** the second index!

```
>>> st = 'Hello!'
>>> li = [1, 2, 'ab', 3.14, 'c']
>>> st[1:4]
'ell'
>>> li[2:4]
['ab', 3.14]
```

  - You can also use **negative indices** when slicing:

```
>>> st[1:-1]
'ello'
>>> li[1:-2]
[2, 'ab']
```

Slicing

```
>>> st = 'Hello!'
>>> li = [1, 2, 'ab', 3.14, 'c']
```

- Omit the first index `[:5]` to make a copy starting from the beginning:

```
>>> st[:4]
'Hell'
>>> li[:3]
[1, 2, 'ab']
```

- Omit the second index `[2:]` to make a copy till the end:

```
>>> st[1:]
'ello!'
>>> li[-3:]
['ab', 3.14, 'c']
```

Copying the whole sequence

- Use [:] to make a ***copy*** of an entire list

```
>>> li = [1, 2, 'ab', 3.14, 'c']
>>> li2 = li[:]
```

- A copy means changes to one does not affect the other!

```
>>> li.reverse()
>>> li
['c', 3.14, 'ab', 2, 1]
>>> li2
[1, 2, 'ab', 3.14, 'c']
```

- Remember how reference works with mutable objects:

```
list2 = list1          # 2 names refer to 1 memory object
                       # --> changing one affects both
list2 = list1[:]       # Two independent objects, two refs
```

The versatile in operator

- For strings, in tests for *substrings*

```
>>> 'bat' in 'combative'
True
>>> 'cat' in 'combative'
False
```

- For lists and tuples, in tests for *membership*

```
>>> medals = ('gold', 'silver', 'bronze')
>>> 'zinc' in medals
False
>>> 'zinc' not in medals
True

>>> li
['c', 3.14, 'ab', 2, 1]
>>> 3.141592 in li
False
```

in is also used as a keyword in the syntax of for loops and others

The + operator

- + produces a *new* tuple, list, or string whose value is the **concatenation** of its arguments.

```
>>> ('Homer', 'Marge') + ('Bart', 'Lisa', 'Maggie')
('Homer', 'Marge', 'Bart', 'Lisa', 'Maggie')
```

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
```

```
>>> 'cat' + 'dog' + 'fox'
'catdogfox'
```

The * operator

- * produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> ['a', 'b', 'c'] *3
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

```
>>> 'Hello' * 3
'HelloHelloHello'
```

String indexing examples

```
>>> w = 'python'
>>> w[2]
't'
>>> w[5]
'n'
>>> w[9]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    w[9]
IndexError: string index out of range
>>> w[-3]
'h'
>>> w[2:4]
'th'
>>> w[0:3]
'pyt'
>>> w[2:]
'thon'
>>> w[-5:]
'ython'
```

p	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

```
>>> w[:4]
'pyth'
>>> w[:-1]
'pytho'
>>> w[2:100]
'thon'
>>> w[:2] + w[2:]
'python'
>>> w[:3] + w[3:]
'python'
>>> w[:]
'python'
>>> w[4:4]
''
```

Practice: *a* vs. *an*

3 minutes

Indefinite NP Generator

- Write a Python <u>script</u> that:
  - prompts for a noun
  - prints out the **indefinite article (a/an)**, and the noun

← It must pick the correct indefinite article!

← "a cat" vs. "an owl"

```
>>> ======================
>>>
What is your noun? cat
a cat
>>> ======================
>>>
What is your noun? owl
an owl
>>>
```

- Describe the algorithm, step-by-step. Specify which Python function you will use with each step.

a_an.py - F:/Portable Python 2.7.3.1/a_an.py

File Edit Format Run Options Windows Help

```
#-----------------------------------------------------------------
# a_an1.py
# Prompts for a noun, and then prints out a/an and then the noun
# Demonstrates string indexing
#-----------------------------------------------------------------

noun = raw_input('What is your noun? ')

if noun.startswith('a') or noun.startswith('e') \
    or noun.startswith('i') or noun.startswith('o') \
    or noun.startswith('u') :
    art = 'an'
else :
    art = 'a'

print art, noun
```

Attempt 1.
Works, but crude

Ln: 18 Col: 0

a_an.py - F:/Portable Python 2.7.3.1/a_an.py

File Edit Format Run Options Windows Help

```
#-----------------------------------------------------------
# a_an2.py
# Prompts for a noun, and then prints out a/an and then the noun
# Demonstrates string indexing
#-----------------------------------------------------------

noun = raw_input('What is your noun? ')

vowels = ['a', 'e', 'i', 'o', 'u']      # list of "vowel"s

if noun[0] in vowels :                  # first char is a vowel
    art = 'an'
else :
    art = 'a'

np = art+' '+noun
print np
```

Attempt 2.
Elegant!
Uses string indexing & list membership test

Ln: 1 Col: 0

Practice: Gerund Generator

3 minutes

- Write a Python script that:
  - prompts for a verb (base form)
  - and then prints out its **gerund form**, i.e., suffixed with **'-ing'**

```
>>>
What is your verb? sleep
sleeping
>>> ========================
>>>
What is your verb? make
making
>>>
```

- CAVEAT: Account for verbal stems that end with **'–e'**
  - walk → walking
  - sleep → sleeping
  - make → making
  - live → living
- KEY: How to derive 'mak' from 'make'?

verb_ing.py - F:/Portable Python 2.7.3.1/verb_ing.py

File Edit Format Run Options Windows Help

```
#-----------------------------------------------------------
# verb_ing.py
# Prompts for a verb, and then prints out the gerund form
#     '-e' will be dropped from the end of verbal stem
#     NOTE: Script will produce be -> *bing, see -> *seing
# Demonstrates string indexing
#-----------------------------------------------------------

verb = raw_input('What is your verb? ')

if verb[-1] == 'e' :                    # make, live, argue
    stem = verb[:-1]                    # mak, liv, argu
else :
    stem = verb

ger = stem+'ing'
print ger
```

Ln: 17 Col: 9

Looking ahead

- `for x in LIST :`
  - iterates through a list for doing something to each element

```
>>> simpsons = ['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie']
>>> for s in simpsons :
        print s, 'is a Simpson.'

Homer is a Simpson.
Marge is a Simpson.
Bart is a Simpson.
Lisa is a Simpson.
Maggie is a Simpson.
>>>
```

Press ↵ **ENTER** at the end of each line

← Iterates through every element `s` of the `simpsons` list and prints the value of `s` followed by 'is a Simpson.' .

Wrap-up

- **Next class**
  - Loops! `for` loop, `while` loop

- **Exercise #4**
  - Due Tuesday midnight
  - Practice Python for 1 hour: review what we learned in class, and explore on your own.