CS 112 Sequences, Lists and Tuples

Dan Fleck
Spring 2009
George Mason University

Coming up: Quick Review for Lab 4

Quick Review for Lab 4

Coming up: Base Numbering Systems

Base Numbering Systems

- Numbers can be represented by and to the computer using various base number systems
- At this point, we are particularly interested in:
  - Binary (base 2)
  - Octal (base 8) # Not in lab 4, but common
  - Decimal (base 10)
  - Hexadecimal (base 16)

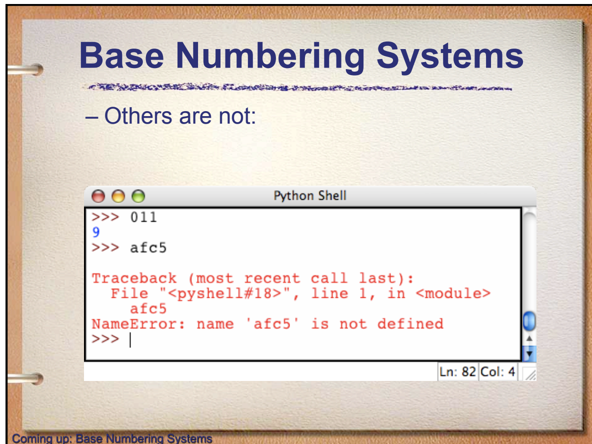
Coming up: Base Numbering Systems

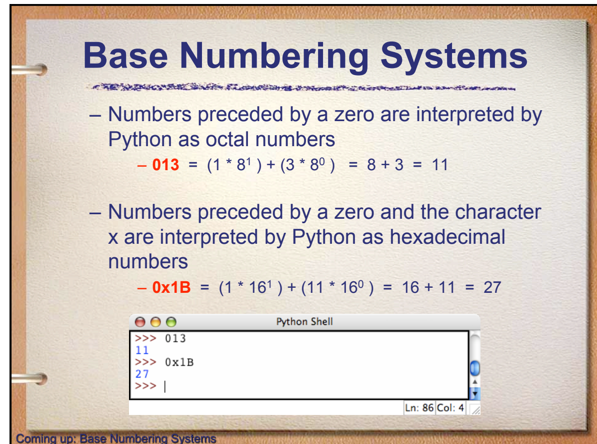
Base Numbering Systems

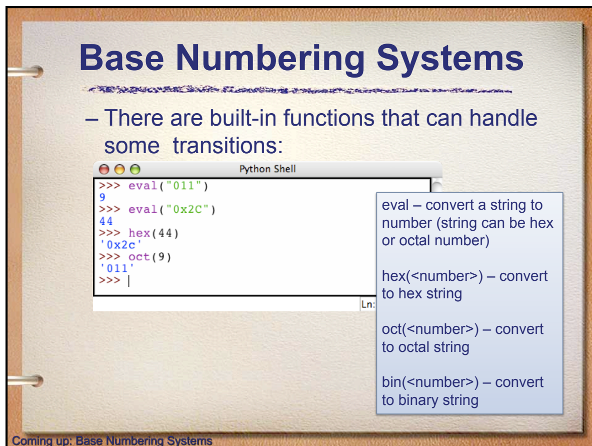
- Python has various mechanisms for handling different number bases
  - Base 10 is recognized implicitly:

```
>>> 10
10
>>> 105
105
>>> 2112
2112
>>>
```

Coming up: Base Numbering Systems







Base Numbering Systems

- Binary
  - digits 0-1
- Octal
  - digits 0-7
- Decimal
  - digits 0-9
- Hexadecimal
  - digits 0-9 & A(10)-F(15)

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

0 0 1 0 1 0 0 1

32 + 8 + 1 = 41

2) 41 r 1
2) 20 r 0
2) 10 r 0
2) 5 r 1
2) 2 r 0
2) 0 r 1

1 0 1 0 0 1

Coming up: String as a Sequence

String as a Sequence

- String: An immutable sequence of characters
  - immutable: cannot be changed
  - sequence: a particular order in which things follow each other
    - forward index ⇒ 0 through n-1
    - backward index ⇒ -1 through -n
  - character: individual ascii symbols

String Sequence

- theStr = 'index'

0	1	2	3	4
i	n	d	e	x
-5	-4	-3	-2	-1

Indexing example

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print greet[0], greet[2], greet[4]
H l o
>>> x = 8
>>> print greet[x - 2]
B
```


Indexing example - from the right

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

String Data Structure

- Immutability:
  - individual elements (characters) can not be changed once created
  - the string can be recreated
  - the variable can be re-defined

Coming up: String Immutability

String Immutability

```
def main():
    theStr = "I am a string"

    # Read a character
    print theStr[3]

    # Attempt to write (modify) a character
    theStr[3] = 't'
```

An attempted String mutation

```
m

Traceback (most recent call last):
  File "/Users/dfleck/Documents/gmuwebsite/classes/cs112/spring09/samplecode/stringSequenceExamples.py", line 14, in <module>
    main()
  File "/Users/dfleck/Documents/gmuwebsite/classes/cs112/spring09/samplecode/stringSequenceExamples.py", line 12, in main
    theStr[3] = 't'
TypeError: 'str' object does not support item assignment
>>>
```

Coming up: String re-creation

String re-creation

- Recreating or reassigning a string is fine:

*stringSequenceExamples.py – /Users/dfleck/Documents/gmu

```
# Strings as a sequence examples

def main():
    theStr = "I am a string"

    # Read a character
    print theStr[3]

    # Attempt to write (modify) a character
    #theStr[3] = 't'

    # Recreate the string is okay
    aNewStr = "I at a string"; print "1:", aNewStr
    aNewStr = theStr.upper(); print "2:", aNewStr

    # Reassign a string variable to a new string is okay
    theStr = aNewStr  ; print "3:",theStr
    theStr = theStr.lower(); print "4:",theStr
```

```
>>>
m
1: I at a string
2: I AM A STRING
3: I AM A STRING
4: i am a string
>>>
```

Coming up: String Methods

String Methods

- Many string methods return a new string (because they cannot modify (mutate) the original string).

- aStr = “hello world”
- bStr = aStr.capitalize() # Does this change aStr?
- aStr = aStr.capitalize() # Is this legal?

Coming up: Sequence Operators

Sequence Operators

3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange

There are six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects.

String literals are written in single or double quotes: `'xyzzy'`, `"frobozz"`. See chapter 2 of the *Python Reference Manual* for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding `"u"` character: `u'abc'`, `u"def"`. Lists are constructed with square brackets, separating items with commas: `[a, b, c]`. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as `a, b, c` or `()`. A single item tuple must have a trailing comma, such as `(d,)`.

Buffer objects are not directly supported by Python syntax, but can be created by calling the builtin function `buffer()`. They don't support concatenation or repetition.

Xrange objects are similar to buffers in that there is no specific syntax to create them, but they are created using the `xrange()` function. They don't support slicing, concatenation or repetition, and using `in`, `not in`, `min()` or `max()` on them is inefficient.

Most sequence types support the following operations. The `"in"` and `"not in"` operations have the same priorities as the comparison operations. The `"+"` and `"*"` operations have the same priority as the corresponding numeric operations.[3.3]

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, *s* and *t* are sequences of the same type; *n*, *i* and *j* are integers:

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)
`s * n , n * s`	*n* shallow copies of *s* concatenated	(2)
`s[i]`	*i*'th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3), (4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3), (5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	

You've already seen these

Coming up: Sequence Operations

Sequence Operations

Python Shell

```
>>>
>>> a_string = "1234567890"
>>>
>>> len(a_string)
10
>>>
>>> "5" in a_string
True
>>>
>>> "A" not in a_string
True
>>> |
```

Ln: 95 Col: 4

Python Shell

```
>>> a_string = "1234567890"
>>> "234" in a_string
True
>>> |
```

Ln: 75 Col: 4

Coming up: What about a substring?

What about a substring?
Slicing a string

- Slicing:
  <string>[<start>:<end>]
- start and end must both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

Coming up: Slicing Example

19

Slicing Example

H	e	l	l	o		B	o	b

0 1 2 3 4 5 6 7 8 9

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

Hint: When slicing it helps to think of the slice indexes between the characters, then 0:3 is very clear

Coming up: String Declaration & Initialization

20

String Declaration & Initialization

- Declaring an Empty String
  - a_string = empty single or double quotes
- Note: nothing inherently special about name (just another identifier) so self-documenting code helps...
  - x = 5   x = “”   first_name = “”

Coming up: MIN & MAX Functions

MIN & MAX Functions

- **min(sequence)**: returns the element in the sequence that has the minimum “value”
- **max(sequence)**: returns the element in the sequence that has the maximum “value”
- Based on ASCII code value for string sequences

Coming up: ORD & CHR Functions

ORD & CHR Functions

- **ord(char):** converts single character to corresponding ASCII integer value
- **chr(int):** converts integer value to corresponding character symbol
- Based on ASCII code value
  - American Standard Code for Information Interchange
  - 7 binary bits ⇒ 128 unique symbols
- Python also supports Unicode (16 bits)

Coming up: ASCII Table

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

American Standard Code for Information Interchange

Coming up: ORD & CHR Functions

ORD & CHR Functions

Python Shell

```
>>>
>>> ord("A")
65
>>>
>>> chr(65)
'A'
>>>
>>> ord("AA")

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    ord("AA")
TypeError: ord() expected a character, but string of length 2 found
>>> |
```

Ln: 31 Col: 4

Coming up: MIN & MAX Functions

MIN & MAX Functions

Python Shell

```
>>>
>>> min("123abc")
'1'
>>> max("123abc")
'c'
>>> min("321cba")
'1'
>>> max("321cba")
'c'
>>> ord("1"); ord("2"); ord("3"); ord("a"); ord("b"); ord("c")
49
50
51
97
98
99
>>> |
```

Ln: 50 Col: 4

Is it alphabetical ordering? Be careful: min('abcWXY') ?

Coming up: String Methods

String Methods

Python Shell

```
>>>
>>> a_string = "this is a sentence."
>>>
>>> new_string = a_string.capitalize()
>>>
>>> new_string
'This is a sentence.'
>>>
>>> a_string
'this is a sentence.'
>>> |
```

Ln: 33 Col: 4

Coming up: Sequence Operators vs. Object Methods

Sequence Operators vs. Object Methods

- **sequence operators**:
  - may be built-in functions
  - operate on various data types
  - example: len   len(a_string)
- **methods**: operate on a single data type
  - string object methods (e.g., capitalize)
  - a_string.capitalize()

Coming up: Comparison Operations

Comparison Operations

Python Shell

```
>>>
>>> string_1 = "A"
>>> string_2 = "B"
>>>
>>> string_1 < string_2
True
>>>
>>> ord("A"); ord("B")
65
66
>>> |
```

Ln: 43 Col: 4

Coming up: Comparison Operations

Comparison Operations

Python Shell

```
>>> string_3 = "ABC"
>>> string_4 = "BCD"
>>>
>>> string_3 > string_4
False
>>>
>>> string_5 = "ABCDEF"
>>>
>>> string_3 > string_5
False
>>> string_3 == string_5
False
>>> string_3 < string_5
True
>>>
>>> string_5 = "ABCDEF"
>>> string_6 = "GH"
>>>
>>> string_5 < string_6
True
>>>
```

Ln: 64 Col: 4

Coming up: Sequence Comparison Operations

Sequence Comparison Operations

untitled.py - /Users/rheishma/112/untitled.py

```
def main():

    string_1 = "abc"; string_2 = "def"; string_3 = "ghi"

    print string_1 > string_2 or string_1 != string_3
    print string_2 == string_3 and  string_1 <= string_2

main()
```

Ln: 8 Col: 6

Python Shell

```
>>> ===================== RESTART =====================
>>>
True
False
>>>
```

Ln: 23 Col: 4

Coming up: Dissecting Data Streams

Dissecting Data Streams

– "FAC50000BC4A01015CC01010"
  » Is there a pattern?

"FAC50000BC4A01015CC01010"

Coming up: Dissecting Data Streams

Dissecting Data Streams

```
def main():
    test_str = "FAC50000BC4A01015CC01010"
    print "Address Data"
    print "------------"
    print " " + test_str[0:4] + \
      "  " + test_str[4:8]
    print " " + test_str[8:12] + \
      "  " + test_str[12:16]
    print " " + test_str[16:20] + \
      "  " + test_str[20:24]

main()
```

```
Address Data
------------
FAC5    0000
BC4A    0101
5CC0    1010
```

Coming up: Terminology / Concepts

Terminology / Concepts

- Binary Number System
- Octal Number System
- Hexadcimal Number System
- ASCII
- Slicing/Substring

Coming up: Tuple and Lists Data Structures

Tuple and Lists Data Structures

- Tuple: An immutable sequence of valid Python data types
- List: A mutable sequence of valid Python data types
- Tuples, Lists and Strings are all Python sequence data types.

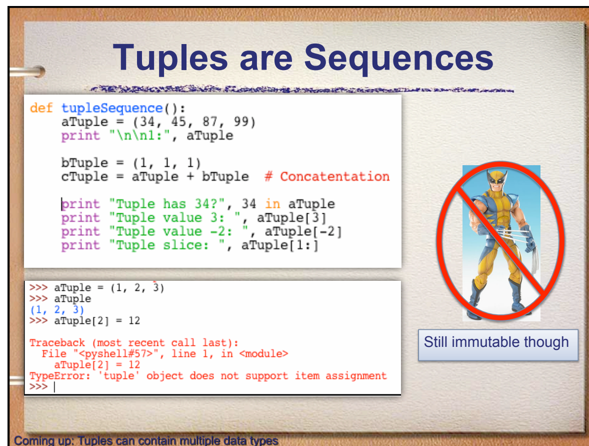
Coming up: Tuple Declaration & Initialization

Tuple Declaration & Initialization

```
>>> animal_tuple = ()                  # Empty tuple
>>> type(animal_tuple)
<type 'tuple'>
>>>
>>> animal_counts = (1, 7, 3, -19)     # Tuple with data
>>> type(animal_counts)
<type 'tuple'>
>>>
>>> animal_age = (34)                  # Oops… what happened?
>>> type(animal_age)
<type 'int'>
>>>
>>> animal_age = (34,) # Gotta have the comma
>>> type(animal_age)
<type 'tuple'>
>>> |
```

- Note: nothing inherently special about name (just another identifier) so self-documenting code helps...
  - x = 5   x = ()   employee_tuple = ()

Coming up: Tuples are Sequences



Tuples can contain multiple data types

```
>>> aTuple = ("Dan", "was", 25, 300.0, "seconds")
>>>
>>> bTuple = ("Something", 35, aTuple)
>>>
```

Tuples can even contain other tuples!

```
>>> print aTuple
('Dan', 'was', 25, 300.0, 'seconds')
>>>
>>> print bTuple
('Something', 35, ('Dan', 'was', 25, 300.0, 'seconds'))
>>> |
```

Coming up: Tuples as Lookup Tables

Tuples as Lookup Tables

- Tuples are frequently used to created lookup tables.
- Requirement: Ask the user for a number and convert that to an appropriate month
- Lets try it!

Coming up: Lists are also sequences

Lists are also sequences

```
>>> aList = []
>>> type(aList)
<type 'list'>
>>>
>>> gradeList = [89, 98, 101, 23]
>>> type(gradeList)
<type 'list'>
>>>
>>> gradeList = [89]
>>> type(gradeList)
<type 'list'>
>>>
>>> |
```

- Lists are exactly like tuples, EXCEPT they are mutable!

Coming up: Lists: Mutable!

Lists: Mutable!

```
>>> gradeList = [89, 98, 101, 23]
>>> gradeList[1] = 22
>>> print gradeList
[89, 22, 101, 23]
>>>
```

Operation	Result
x in s	True if an item of *s* is equal to *x*, else False
x not in s	False if an item of *s* is equal to *x*, else True
s + t	the concatenation of *s* and *t*
s * n, n * s	*n* shallow copies of *s* concatenated
s[i]	*i*'th item of *s*, origin 0
s[i:j]	slice of *s* from *i* to *j*
s[i:j:k]	slice of *s* from *i* to *j* with step *k*
len(s)	length of *s*
min(s)	smallest item of *s*
max(s)	largest item of *s*

These all work with Strings, Tuples and Lists

Coming up: Whoa there cowboy…

Whoa there cowboy…

- Shouldn't there be some special operators that work with mutable data structures like lists?
- Yep --- wait till next week!
- http://docs.python.org/library/stdtypes.html#mutable-sequence-types

Coming up: String vs. Tuple vs. List

String vs. Tuple vs. List

- String
  - sequence of characters only
  - immutable data structure
- Tuple
  - sequence of valid Python data types
  - immutable data structure
- List
  - sequence of valid Python data types
  - mutable data structure

Coming up: Note on Tuple Initialization

Note on Tuple Initialization

tuples.py - /Users/rheishma/112/tuples.py

```
def main():

    a_tuple = ()
    print a_tuple

    tuple_x = 12,23,34,45
    print tuple_x

    tuple_y = (12,23,34,45)
    print tuple_y

main()
```

Ln: 10 Col: 17

This works, but is confusing… use parens!

Python Shell

```
>>>
()
(12, 23, 34, 45)
(12, 23, 34, 45)
>>>
```

Ln: 97 Col: 4

Coming up: Accessing Tuple Elements

Accessing Tuple Elements

tuples.py – /Users/rheishma/python/tuples.py

```
def main():
    a_tuple = (23, 45, "box", 12.5, (1,2,3));
    print a_tuple[1], "*", a_tuple[2], "*", a_tuple[4]
    print a_tuple[2][0], "*", a_tuple[4][2]
main()
```

Ln: 12 Col: 0

Python Shell

```
>>> ====================== RESTART ======================
>>>
45 * box * (1, 2, 3)
b * 3
>>> |
```

Ln: 27 Col: 4

Coming up: Repetition & Concatenation

Repetition & Concatenation

tuples.py – /Users/rheishma/python/tuples.py

```
def main():
    a_tuple = (23,"box", 12.5, (1,2,3)); print a_tuple
    a_tuple *= 2; print a_tuple
    a_tuple += (5,6); print a_tuple
    a_tuple += (5,6),; print a_tuple
main()
```

Must be careful with syntax…
* attention to detail *

Ln: 14 Col: 0

Python Shell

```
>>> ====================== RESTART ======================
>>>
(23, 'box', 12.5, (1, 2, 3))
(23, 'box', 12.5, (1, 2, 3), 23, 'box', 12.5, (1, 2, 3))
(23, 'box', 12.5, (1, 2, 3), 23, 'box', 12.5, (1, 2, 3), 5, 6)
(23, 'box', 12.5, (1, 2, 3), 23, 'box', 12.5, (1, 2, 3), 5, 6, (5, 6))
>>>
```

Ln: 56 Col: 4

Coming up: Modifying Tuple Elements

Modifying Tuple Elements

tuples.py – /Users/rheishma/112/tuples.py

```
def main():

    a_list = [1,2,3]
    a_tuple = ("box", 12.5, a_list)
    print a_tuple
    a_list[1] = 5
    print a_tuple
main()
```

While a tuple may be immutable, tuple elements may contain imbedded references to mutable data types

Ln: 9 Col: 7

Python Shell

```
>>> ====================== RESTART ======================
>>>
('box', 12.5, [1, 2, 3])
('box', 12.5, [1, 5, 3])
>>>
```

Ln: 182 Col: 4

Coming up: Practice with slicing/indexing

Practice with slicing/indexing

- Slicing and indexing is critical to know

- aString = 'abc123'
- aString [2]
- aString [0:4]
- aString [-2]
- aString [3:]

Coming up: Practice with slicing/indexing

Practice with slicing/indexing

- Slicing and indexing is critical to know
- a= [‘abc’, 123, ‘ddd’, 999]
- a[2]
- a[0:4]
- a[-2]
- a[1:]

Coming up: Formatted Printing

Formatted Printing

untitled.py - /Users/rheishma/112/untitled.py

```
def main():

    print "%x" % 108
    print "%#x" % 108

    x = "%.2f" % 1234.56789
    print x

    print "%e" % 1234.56789

main()
```

Ln: 6 Col: 0

Python Shell

```
>>>
6c
0x6c
1234.57
1.234568e+03
>>>
>>> 6c
SyntaxError: invalid syntax
>>> 0x6c
108
>>>
```

Ln: 426 Col: 4

Coming up: String Justification

String Justification

- Justification
  - aligns text within a text field in a particular fashion:
    - left justified - “text “
    - center justified - “ text “
    - right justified - “ text“
  - print “text”.ljust(20)
  - print “text”.center(20)
  - print “text”.rjust(20)

Coming up: Terminology / Concepts

Terminology / Concepts

- Indefinite Loop
- Definite Loop
- Mutable / Immutable
- Sequence
- String Data Type
- Tuple Data Type
- Justification

52

End of presentation