Intro to Python Notes #3
Basic Python Data Types

Downey Book

- This is a time-limited "breadth-first" approach to learning Python
  - Covers major points such as data types, operators, if and for statements without going into a great deal of depth
- We've already covered a bit of material from Ch 3, 5, 7
- What follows is great deal of Downey Ch.2

Sweigart Book

- If you are new to programming you may find Sweigart's first few chapters to be a useful and gentle introduction

Integers

- Numbers without a decimal point, positive or negative
- Conceptually unlimited in Python 3
- Many other languages limit integers to a certain size, e.g, 2^{64}-1 or 2^{32}-1
  - Python 2 has limited integers and longs

Floating Point Numbers

- Aka Floats, Singles, Doubles
- Floats can be written two ways:
  - fixed decimal numbers `12.567`
  - exponential (scientific) notation `1.2567e1`
- Python uses the two formats interchangeably and you don't have to worry about them
- There are ways to force Python to display the numbers in one way or the other

Floats

- Floating point numbers have a maximum number of **significant digits** and can never exceed that number of digits
- Floats are often **rounded** and so are not precise
- For example, if you divide 1 by 3 you get `0.3333333333333333` which is not the "correct answer" but close enough

Floating point arithmetic

- Computation with floats is imprecise
  - Many of the normal laws of arithmetic may not hold 100% of the time
  - For example, if x = a/b, it is not necessarily true that b*x = a
  - It may not be true that (a+b)+c = a+(b+c)
- A branch of computing called numerical analysis is devoted to the science of computing accurately with numbers

Internal Representations

- A reminder from last set of notes. Strings and ints are "precise" representations. Note the repeating pattern 1001 in 0.2. It never terminates
- We see the same in decimal 1/3 = 0.3333333….

```
2       00000000 00000000 0000000 00000010
'2'     00110010
2.0     01000000 00000000 0000000 00000000
-2      11111111 11111111 11111111 1111110
'-2'    00101101 00110010
-2.0    11000000 00000000 0000000 00000000
'0.2'   00111000 00101110 00110010
0.2     00111110 10011001 10011001 10011001 …
```

The type() function

- When you are working in the Python interpreter, you can use the type() function to determine the type of an object
- English ⇔ Python

Integer	int
Float	float
String	str

```
>>> h = 12345678901234567890
>>> type (h)
<class 'int'>
>>> k = 1.0e50
>>> type(k)
<class 'float'>
>>> l = 1+3j
>>> type(l)
<class 'complex'>
>>> q = """asb
asfsasdf
asdfadf"""
>>> type (q)
<class 'str'>
```

Comparing Types

- Note that the type() function returns a type, not a string. To compare types, compare the results of the type() function!
-

```
>>> a = 3
>>> type(a)
<class 'int'>
>>> type(a) == "<class 'int'>"
False
>>> type(a) == type(1)
True
>>>
```

Mixed Types in Expressions

- You can mix ints and floats in arithmetic expressions
- When you do, all the numbers involved are converted to floats and the expression is computed
- The result is a float unless you use some function to convert the result to an int or something else like a string
- The standard arithmetic operators are +, -, *, /, ** and %

+,-,*, / Operators

- +, -, * behave (more or less) as you might expect
  - In Python 2 the division operator / with ints produces an integer result

```
7/3 = 2          8/3 = 2
```

- In Python 3 division with ints produces a float

```
>>> 7/3
2.3333333333333335
>>> type(7/3)
<class 'float'>
>>> 9/3
3.0
>>> type (9/3)
<class 'float'>
```

% Operator

- % is the modulus (remainder after division) operator

```
>>> 7%3
1
>>> 8 % 3
2
>>> -7%3
2
>>> 4.22 % 3
1.2199999999999998
```

- Note that -3 * -3 + - 2 = 7 and note the imprecise result of 4.22%3

** Operator

- ** is exponentiation (raising to a power)

```
>>> 3 ** 3
27
>>> 27 ** (1/3)
3.0
>>> 2 ** 5
32
>>> 2**77
151115727451828646838272
>>> 144**0.5
12.0
```

Characters

- These are the single characters that you type as well as some that you can't see on the screen such as Enter (newline), tab or backspace
- Strings are not an "atomic" type since you can break them into characters
- Each character has a number
  - for Python 2.x this number runs from 0 to 255
  - Only characters with numbers in the range 32…126 display on the screen

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Standard ASCII – Pretty much the same on all computers worldwide

Question

- What are the ASCII codes for 'Hello,World!'

Answer

```
for c in "'Hello,World!'":
 print c,ord(c)
```

```
' 39
H 72
e 101
l 108
l 108
o 111
, 44
 32
W 87
o 111
r 114
l 108
d 100
! 33
' 39
```

chr and ord

- You can create character n by using the Python function chr(n)
- Thus chr(70) = 'F'
- The reverse function is called ord() – it takes a character and gives its number
- Thus, ord('F') = 70

Examples

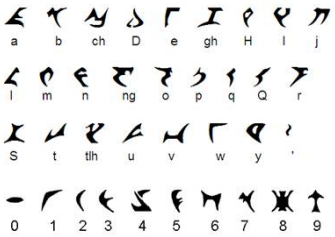
Expression	Value	Expression	Value	Expression	Value	Expression	Value
ord('g')	103	ord('G')	71	ord('a')	97	ord('A')	65
chr(80)	P	chr(83)	S	chr(90)	Z	chr(38)	&
ord('?')	63	ord('*')	42	ord('/')	47	ord('.')	46
chr(45)	-	chr(92)	\	chr(95)	_	chr(74)	J

Unicode

- In Python 2.xx, chr() and ord() only work with the Extended ASCII characters
  - Uses only numbers in range(256) = [0, 1,…,255]
- In Python 3.x, chr() and ord() work with Unicode – a giant alphabet that pretty much includes all characters in all languages
- Dealing with Unicode (and character encodings) is a complex topic that we have not enough time to discuss well, but …

Klingon!

- Unicode even includes some languages that don't exist, such as Klingon



Unicode

```
 <<<chr(65)
'A'
>>> print (chr(65))
A
>>> chr(165)
'¥'
>>> chr(1665)
'ځ'
>>> chr(16665)
'禖'
>>> chr(166665)
'\U00028b09'
```

Boolean Values

- They are **`True`** and **`False`**
  - Any other variation are not Boolean values
  - **`true, TRUE, tRuE,`** etc are not **`True`**
  - **`false, FALSE, fAlSe,`** etc are not **`False`**
- Comparisons produce Boolean values

```
>>> a = 1
>>> b = 2
>>> c = a == b
>>> print (c)
False
```

Boolean Variables

- Variables can hold Boolean values

```
won = myScore > 1000
if won
  print ("You won!")
else
  print ('Sorry! You lost everything!')
```

Sequences

- Python provides several sequence types, including
  - strings (str)
  - lists
  - tuples
- And a few others that we won't mention just yet
- Many of the properties and operations that strings have are also possessed by the other sequences

Sequences

- A string is a sequence of characters
  - Designated by matching quote marks (' or ")
  - Cannot be modified (immutable)
- A list is a sequence of objects
  - Designated by square brackets
  - Can be modified (mutable)

```
[1, 2, '1', [3, 4], 'Hello']
```

- A tuple is like a list, but cannot be modified
  - Designated by parentheses

```
(1, 2, '1', (3, 4), 'Hello')
```

Strings

- A string is a sequence of characters
- You can access the characters one by one as follows

```
st = "Hello, World!"
Here st[0] = "H", st[1] = "e",
…, st[12] = "!"
```

- The number inside the square brackets is called the index
- The operation is called indexing
- **We always start the numbering at 0!**

Indexing

- The *indexing* operation is available in Python sequences such as strings or lists
  - Many other languages provide a structure called an array with similar operations
- The positions in a sequence are numbered from the left, starting with 0.
- The general syntactic form is

  `<sequence>[<expr>]`

  where the value of **`<expr>`** determines which character (object) is selected from the string (sequence)

The String Data Type

- The most common uses of personal computers are word processing and text manipulation.
  - Web pages are just long strings
  - Controls in web pages (e.g., a dropdown list) are just long strings
  - Even images can be expressed as long strings (SVG or PNG base64)
- A string value or string literal is a sequence of characters enclosed within quotation marks (") or apostrophes (').

PNG Image


```

9hAAAC60lEQVQ4T11TXUhTYRh+vrPNbWe/jk0EoywotCjILhSlHxRclEXkRQWrLo
w0FAm17uoiiVCcWYghelEpSpEkEsREiIg0kkCctbyIxNBM085+nM79fL1nM5198H
E47/s9z/u8fwz/HR6WirFwz47I13wo57IBBvAMD9jeYdhqXUxlHkyGkDdxOOeZkA
aq4Gsrh+a7HioFoFACgkBOuhF6FM4MQFvWDsPJVsbYlIyLE8TBvzvr4b/pgFELKA
3kkV1EIhMI61+sAf4QoLrdDbPjlkySIFh82YTFS7WwWOmPoiK2CWLrYCYTpdAliH
cJ0D50MuOJOsZXZ4oxfb4Pugm9tGyG3WHDmxcLEHUEjIPoEonkF2A/q8Wb1xyiuA
JIOwJI7y9l/Fu1E7ynBqISHb0mjLrVyNkfRcXlYFIKAjq61BgdUyDnAEfFvVIZlI
i8sZnxyaIRpIzlQalAbmkm+jtm0fDIgpY73o3ospLc42b0dy+j4YEaLY0ciMr1KP
zA+OedEkTJBEVC6j/JcfmybT2F5HQSNhK4lOZlfNwmQbdmgpKMWipeVJPoQEwAlw
mJhMkAua2MpGsp+opcaHqzqCMCd+4I13zJi5LzvTsNR/NmSZ0Yr7YgJOYgJqshv0
IMwTVkROER4hOoncuHKQXPNWcs+rSGWYD23n1QkYCyixMUheaBBMXVqKh9xjCePM
6CzxtCVaUX3OsneysVUW7jZElfTHTrBZMa9zu3wbe6CyVFHqTbdHGC+aUQXg1lwa
CdwvVqH3gwCubbHUDGYGlikH49a+LT52p5qhEC1eHtpz1wvdtO6jWgjOkbRlHBDx
wrmEFkLQWC/w9g7XIqzKfqNkd5pq0ec5UOblCCmUlyRIVYiBaAaiBoKS81BQpQ5G
AYSG3uhvXK5ihv7MP88yrM3S2PYVwPjZomV+6IArFoGCwcoWwOBmC90Q7Lma3LlL
yetFjF+Nljh3cgH8GP2fGNFA95YDg9jPQLLlqgLev8F/c4CyXKIh7+AAAAAElFTk
SuQmCC
```


The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print (str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

A String as a Sequence

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
```

Negative Index Values

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'
```

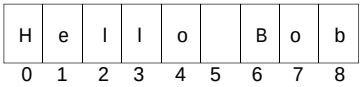
Indexing and Slicing

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.
  - Slicing can also return a single character but indexing can never return a substring
  - Slicing can also return non-contiguous sequences of characters

Slicing Syntax

- Slicing:
  <string>[<start>:<end>]
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position **<end>**.
  - This is a bit counter-intuitive and is a good example of an easy way to create a "boundary error"
- If either expression is missing, then the start or the end of the string are used.

Slicing Examples




```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
>>> greet[6:7]
'B'
```

Examples

```
F = 'watermelon'
F[3:8] = 'ermel'
F[-4:-2] = 'el'
F[:5] = 'water'
F[5:] = 'melon'
F[::-1]
  ='nolemretaw'
```

Negative Indices

- In slice expressions, negative numbers indicate right to left from the end of the string

```
greet = "Hello Bob"
# The last two characters
>>> greet[-2:]
'ob'
#Everything except the last two chars
>>> greet[:-2] 'Hello B'
```

Slicing and …?

- So slicing suggests "taking apart"
  - Can we put two strings together into a longer string? Yes ….
- *Concatenation* "glues" two strings together (+)
- *Repetition* builds up a string by multiple concatenations of a string with itself (*)

Concatenation and Repetition

- Concatenation and Repetition

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The **len** function

- The built-in function **len** gives the length of a string or other sequences as we will see
- **len("Hello, World!")** = 13
- Remember that the length of a string is always one more than the last index since indices start at 0
  - The last character of a string s can be obtained with the expression **s[len(s)-1]**
  - But **s[-1]** is much easier!

Exercise

- What is **len("COS 125")**?
- **7**
- If **A = "Hello, World!"**, what is **A[-4]**?
- **"r"**
- What is **len("COS 125"+A)**?
- **20**
- If **B = "COS 125" + A,** what is **B[-15]**?
- **"2"**
- What is **A[0] + A[7]**?
- **"HW"**

The Empty String

- The ***empty string*** is the only string of length 0
- It is commonly written as "" or ''
- Do not confuse two consecutive single quotes with one double quote
- Note that " " and ' ' are not the empty string, they are the one character string consisting of a single space character

Slicing with an Increment

- AKA "Extended Slicing"

st[[start]:[stop+1][:step]]

- What does this mean?
  - If st is a string, you can select some substring by picking a starting point
  - one more than the stopping point (like range)
  - and an increment (step) if you want one

Examples

```
>>> s = "The quick brown fox jumped over a lazy dog"
>>> s[::2]
'Teqikbonfxjme vralz o'
>>> s[0:13:3]
'T i o'
>>> s[-15::3]
'or zd'
>>> s[:-20:2]
'Teqikbonfxj'
>>> s[::-1]
'god yzal a revo depmuj xof nworb kciuq ehT'
>>>
```

String Operation Summary

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
<string>[::]	Extended Slicing
len(<string>)	Length
For <var> in <string>	Iteration through characters

All of these operators also work with lists and tuples

Strings are Immutable

- That means that you cannot change a string once you define it
- In particular, the following is an error

```
H = 'hello'
H[0] = 'j'
```

So how do you change character 0?

```
>>> st = "Hello"
>>> st[0] = "J"
Traceback (most recent call last):
  File "<pyshell#147>", line 1, in
  <module>
    st[0]="J"
TypeError: 'str' object does not support
  item assignment
>>> st = "J"+st[1:]
>>> st
'Jello'
```

And this works ...

- Each concatenation creates a new string

```
>>> s = ""
>>> for i in range(10):
  s += chr(ord('a')+i)
>>> s
'abcdefghij'
>>>
```

Strings and Lists

- It turns out that strings are really a special kind of *sequence*, so these operations also apply to lists!

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
>>> grades = ['A', 'B', 'C', 'D', 'F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

Strings and Lists

- Strings are always sequences of characters, but *lists* can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 4, "U"]
```

- Unlike strings, lists can contain other lists:

```
myList = [1, 3.14, [1, 2] ]
```

Lists of Lists

- A list can be a multi-dimensional structure composed of other lists
- This could be a tic-tac-toe board

```
>>> board = [[1,2,3],[4,5,6],[7,8,9]]
>>> board
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> board[1]
[4, 5, 6]
>>> board[1][2]
6
```

The Empty List

- Just as we have an empty string we can have an empty list

```
>>> list = []
>>> len(list)
0
>>> list = [[[]]]
>>> len(list)
1
>>> list[0]
[[]]
>>> list[0][0]
[]
>>>
```

range() Syntax

- Note the different numbers of parameters

```
range ([start],limit,[step])
```

- Limit is never reached
- [start] optional - default start is 0
- [step] optional - default step is 1

range() Acts like a List

```
>>> range(10)
range(0, 10)
>>> type (range(10))
<class 'range'>
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,5))
[1, 2, 3, 4]
>>> evens = list(range(2,17,2))
>>> evens
[2, 4, 6, 8, 10, 12, 14, 16]
>>> for i in evens:
   print (i, i**0.5)
>>> for i in range(2,17,2):
   print (i,i**0.5)
```

Tuples

- Tuples (pron. 'tuh-pull') are similar to lists, but are not mutable
  - Syntax: use ( ) instead of [ ]
  - Note that [ ] is the index operator for ALL sequence types

```
t = ("This","is","a","tuple",1,2.2, [(1,2),("abc")])
>>> t
('This', 'is', 'a', 'tuple', 1, 2.2, [(1, 2), 'abc'])
>>> t[6]
[(1, 2), 'abc']
>>> t[6][1][0]
'a'
>>> t[6][0][1]
2
```

Sequences

- Tuples are closely related to "rows" in a relational database
  - We might cover this topic near the end of the semester
- Tuples and lists are two more examples of sequence data types.
  - Will have more to say about sequences later in the course; there are others
- Index numbering for all sequence types start at 0
- Any sequence type can use negative indices as we have indicated elsewhere

Implied Line Continuation

- In writing programs and in the shell you can use new lines inside [] or () without the line continuation character

```
>>> matrix = [
     [1,0,0],
     [0,1,0],
     [0,0,1]]
>>> matrix
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>>
```