Lecture 2: Conditionals, Functions, Strings, Lists, & Loops

Finlay Maguire (finlay.maguire@dal.ca)

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Schedule tweaks: let’s get through the basics so we can get to more fun stuff

Next week: Modules, Notebooks & Reproducible Research
Week after: Functional programming

Overview

- Booleans and conditionals to enable “branching”
- Introduction to functions
- Strings (encoding, formatting, escaping, multi-line, indexing/slicing)
- Lists (creating, lists of lists, accessing elements in a list or a string, slices)
- Tuples and Mutability
- Aliasing vs Copying
- For loops (defining, break, continue, range, zip)

Boolean Types are True or False

Relational Operators
== is equal to
!= is not equal to
> is greater than
< is less than
>= is greater than or equal to
<= is less than or equal to

```
>>> 3 == 1+2
True
>>> 1+2 == 3
True
>>> 42 == "spam"
False
```

```
>>> 42 > 5
True
>>> “A” != “G”
True
>>> not “A” == “G”
True
```

Operations can be chained:

```
x = 4
3 < x and x < 7  == 3 < x < 7
```

Booleans have special operators (cast to integers otherwise)

```
>>> True == 1
True
>>> False == 0
True
>>> True + True
2
>>> True + False
1
>>> True * False
0
```

Boolean Operators

- **and**: True if both are True
- **or**: True if at least one is True
- **not**: True if argument is False

```
>>> True and False
False
>>> False or False or False
False
>>> True and not False
True
```

Booleans have some special functions

```
>>> any((True, False, True))
True

>>> any((False, False))

False

>>> all((True, True, True))

True

>>> all((True, False, True))

False
```

any(L) checks if at least one is True

all(L) checks if all are true

Why are booleans useful? They enable branching!

Booleans enable **conditional** execution

Code so far has been a simple recipe:

do assignment 1

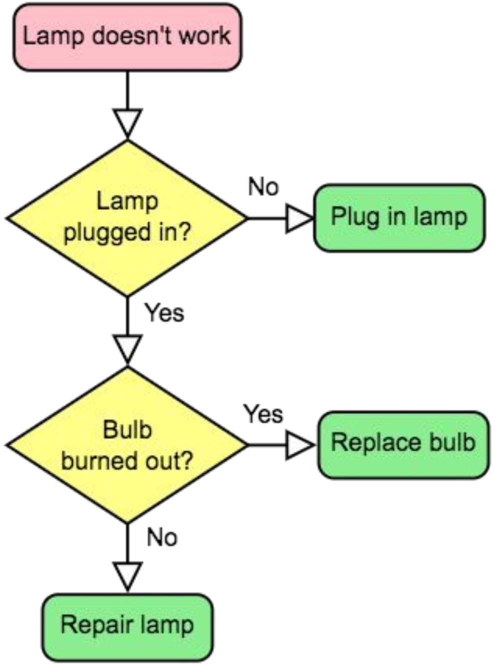
do assignment 2

do assignment 3

…

pass course

Real-world/problems more complex:



https://betterprogramming.pub/5-alternatives-to-if-statements-for-conditional-branching-6e8e6e97430b

Conditionals: boolean expressions and **if**

```
x = 42 * 101
if x == 4242:
    print(“My Office”)
```

“My office”

```
x = 42 * 102
if x == 4242:
    print(“My Office”)
```

…

Iff condition is true then do the code in the “body”.

Body in python is delineated with a **:** (colon) and a “**whitespace**” indentation

Major “gotcha” in python is messing up this whitespace

```
if CONDITION:
    BODY1
```

Conditionals: more than 1 option **if** and **else**

```
x = 4243

if x == 4242:

    print(“My Office”)

else:

    print(“Not mine”)
```

```
“Not mine”
```

If condition is true then run the code in the **BODY1** otherwise run the code in **BODY2**.

Whitespace is still (and in python always will be) important

```
if CONDITION:

    BODY1

else:

    BODY2
```

Conditionals: more than 2 options: **if**, **elif**, and **else**

```
x = 4243

if x == 4242:

    print(“My Office”)

elif x == 4243:

    print(“Old Office”)

else:

    print(“Not mine”)
```

```
“Old Office”
```

If CONDITION1 is true then run **BODY1**, otherwise if **CONDITION2** is true run **BODY2** otherwise run **BODY3**.

```
if CONDITION1:

    BODY1

elif CONDITION2:

    BODY2

else:

    BODY2
```

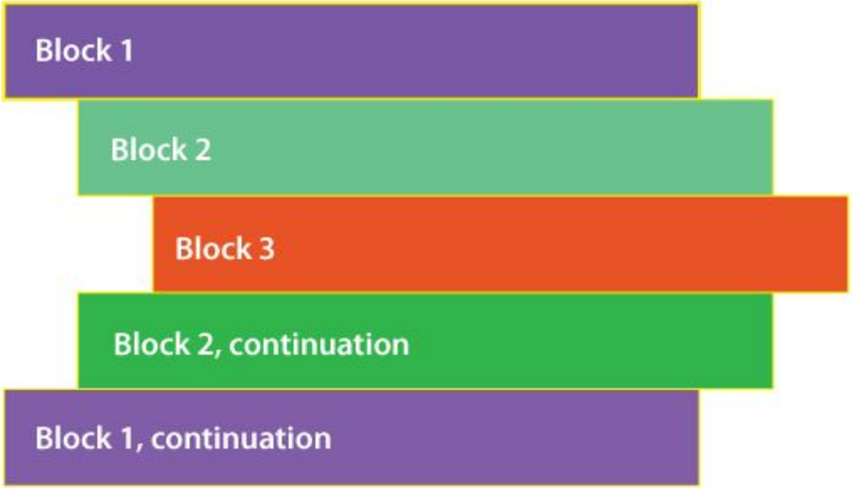
Conditions: order matters.

```
hour = 11
```

```
if hour >= 2 and hour <= 9:
    print("Sleep")
elif hour <= 17:
    print("In class")
elif hour <= 20:
    print("Hang out")
else:
    print("Do Assignment")
```

```
if hour >= 2 and hour <= 9:
    print("Sleep")
elif hour <= 20:
    print("Hang out")
elif hour <= 17:
    print("In class")
else:
    print("Do Assignment")
```

Indentation (& Tabs vs Spaces in Python)



Block 1

Block 2

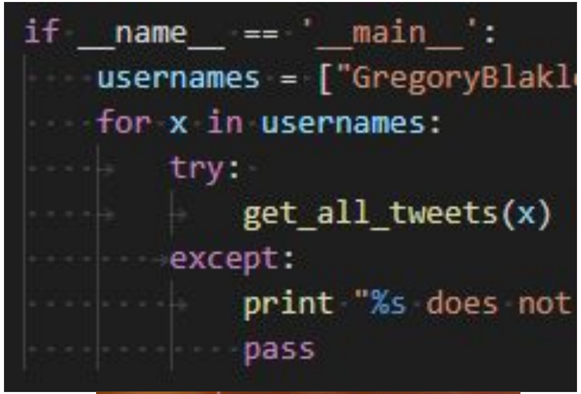
Block 3

Block 2, continuation

Block 1, continuation

PEP8: 4 spaces per indentation

Be consistent or you will get errors

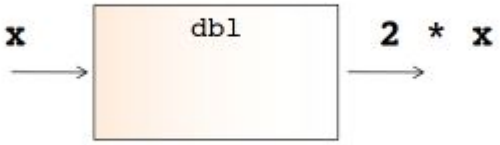


```
if __name__ == '__main__':
    usernames = ["GregoryBlakl
    for x in usernames:
        try:
            get_all_tweets(x)
        except:
            print "%s does not
            pass
```

Branching means lots of repeated code UNLESS we define and use functions

Conditionals and functions use similar syntax

```
def dbl(x):
    return 2 * x

>>> dbl(5)
10
```



```
def dbl(input_val):
    y = 2 * input_val
    return input_val
```

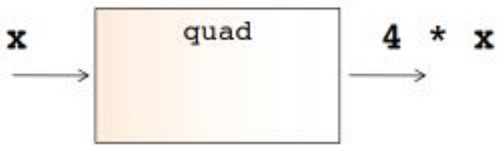
```
def function_name(parameters):
    function body
    return
```

Docstrings are important parts of functions

```
def dbl(x):
    “””This function takes a number
    x as input and returns 2 * 2”””
    return 2 * x
```

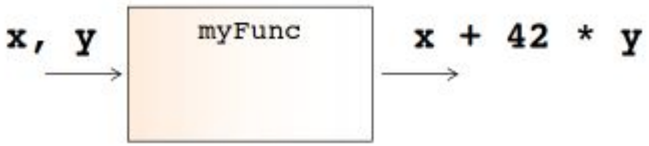
Functions can call other functions

```
def quad(x):
    return 4 * x


def quad(x):
    return dbl(dbl(x))
```



Functions can have multiple inputs

```
def myFunc(x, y):
    “““Returns x + 42 * y”””
    return x + 42 * y
```



Let’s talk about strings a bit more:
Strings are useful and modern python hides a lot of complexity

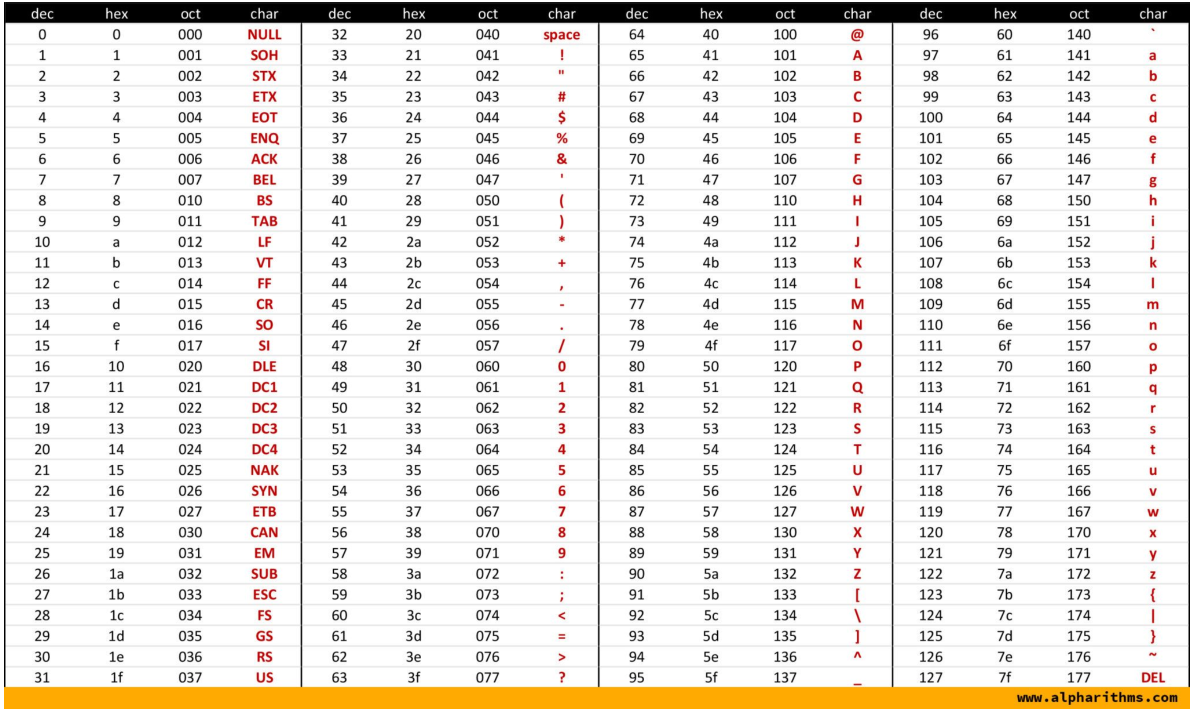
Computers only do numbers -> how does text work?

Text is **encoded** as a number.

ASCII table (128 options)

More characters => more numbers

Unicode v16 (154,998 options) - python uses UTF-8 by default



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

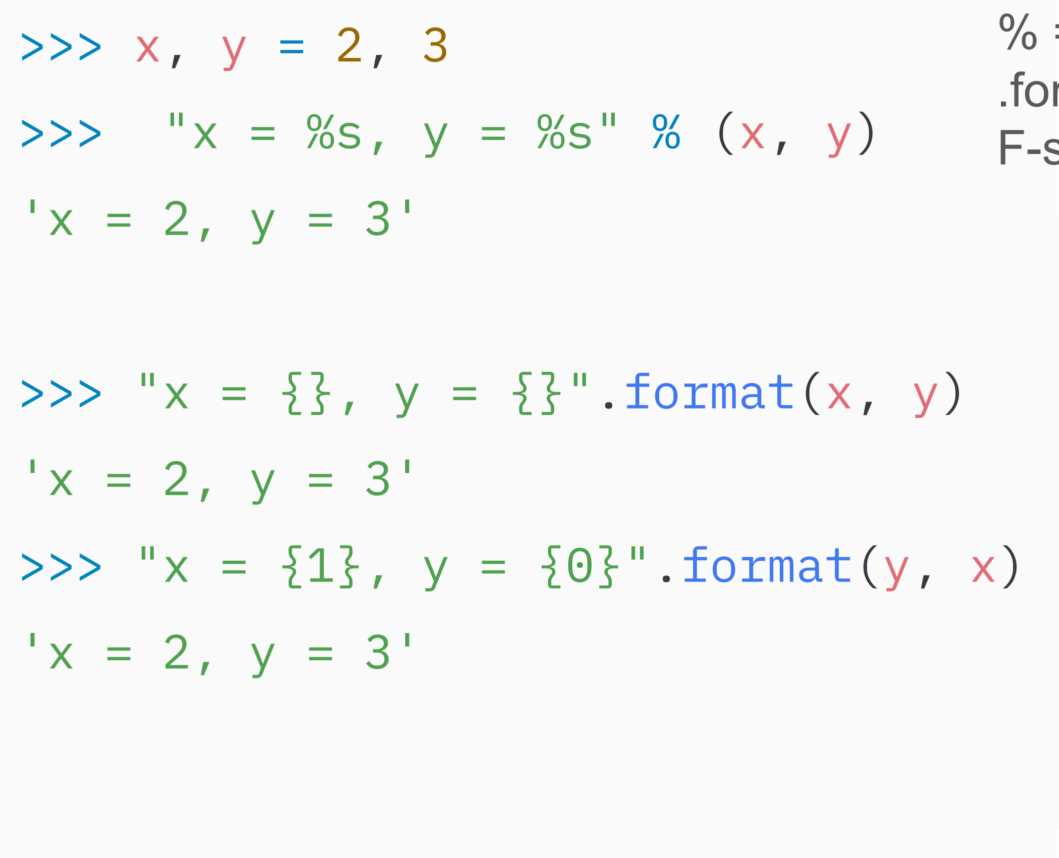
www.alpharithms.com

String additions: concatenation

```
>>> food = "spam"
>>> food
'spam'
>>> food + "!!!"
'spam!!!'
>>> food
'spam'
```

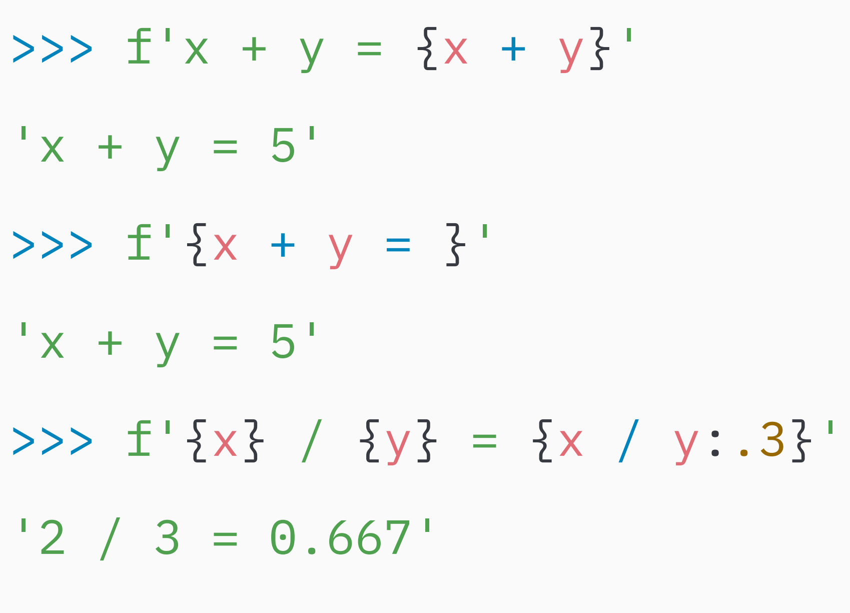
```
>>> food = food + "ityspam"
>>> food
'spamityspam'
```

String formatting/interpolation



```
>>> x, y = 2, 3
>>>  "x = %s, y = %s" % (x, y)
'x = 2, y = 3'


>>> "x = {}, y = {}".format(x, y)
'x = 2, y = 3'
>>> "x = {1}, y = {0}".format(y, x)
'x = 2, y = 3'
```

% = old way likely to be remove
.format = newer way
F-string = newest & cleaner



```
>>> f'x + y = {x + y}'
'x + y = 5'
>>> f'{x + y = }'
'x + y = 5'
>>> f'{x} / {y} = {x / y:.3}'
'2 / 3 = 0.667'
```

Special characters and escaping them

```
>>> print(“a” + “b”)
ab
>>> print(“a” + “\n” +
“b”)
a
b
>>> print(“a\nb\nc”)
a
b
c
```

```
>>> print(“a \\n a”)
a \n b

>>> print(“a \’ in str”)
a ’ in str

>>> print(f”{1+2} and {{“)
3 and {
```

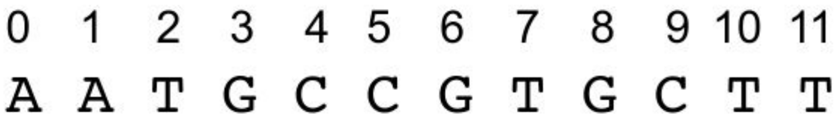
Many built-in string operations

```
> s = 'this is a
string'
> s.capitalize()
'This is a string'
> s.title()
'This Is A String'
> s.upper()
'THIS IS A STRING'
> s.count('i')
3
```

```
> s.title().swapcase()
'tHIS iS a sTRING
> s.removeprefix('this is ')
'a string'
> s.removesuffix(' string')
'this is a'
> s.replace('is', 'IS')
'thIS IS a string'
```

Using strings: length and index

```
>>> dna_seq = "AATGCCGTGCTT"

>>> len(dna_seq)

12

>>> dna_seq[0]

'A'

>>> dna_seq[3]

'G'

>>> dna_seq[20]

IndexError: string index out
of range
```



0	1	2	3	4	5	6	7	8	9	10	11
A	A	T	G	C	C	G	T	G	C	T	T

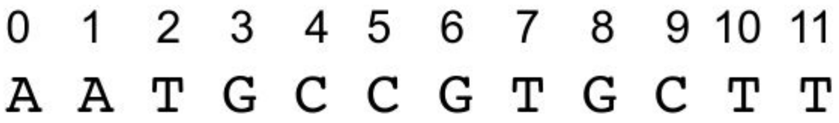
First element in a string is at the 0 position - dna_seq points at a bit of memory and then the index is "**offset**" in memory

string[index]

https://www.cs.hmc.edu/~cs5grad/cs6/slides2021/lec0_2021.pdf

Using strings: length and index

```
>>> dna_seq = "AATGCCGTGCTT"
>>> dna_seq[0:4]
'AATG'
>>> dna_seq[3:7]
'GCCG'
>>> dna_seq[1:]
'ATGCCGTGCTT'
>>> dna_seq[:4]
'AATG'
>>> dna_seq[10:42]
'TT'
```



0	1	2	3	4	5	6	7	8	9	10	11
A	A	T	G	C	C	G	T	G	C	T	T

string[start : stop]

start is just index (inclusive)
stop is a **<** not **<=** (exclusive)

“from start up to stop”
not
“from start up to and including stop”

`string[4] == string[4:5]`

https://www.cs.hmc.edu/~cs5grad/cs6/slides2021/lec0_2021.pdf

Indexing and slicing: negative indices

```
             #           111
             #  012345689012
>>> alphabet = "abcdefghijkl"
>>> alphabet[1:9:3]
'beh'
>>> alphabet[5:0:-1]
'fedcb'
```

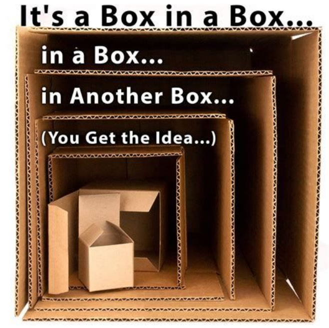
string[**start** : **stop** : **increment**]

from **start** up to **stop** by **increment**

`string[2:6] == string[2:6:1]`

Strings are just a list of characters

Lists are an ordered collection of data



```
primes = [2,3,5,7,11]

biologists = [”McClintock", ”Blackburn", ”Franklin"]

# lists can contain multiple types

L = [2, "turtle", 11]

# lists can include lists

>>> M = [2, "turtle", 11, ["spam", "spamity", "spam"] ]
```

Explicitly converting a list to a string

```
>>> x = “this is a string”

>>> list(x)

[‘t’, ‘h’, ‘i’, ‘s’, ‘ ‘, ‘i’, ‘s’, ‘’, ‘a’, ‘’, ‘s’, ‘t’,
‘r’, ‘i’, ‘n’, ‘g’]

>>> x.split()

[“this”, “is”, “a”, “string”]

>>> x.split(‘a’)

['this is ', ' string']
```

Indexing and slicing the same as strings

```
         0      1      2               3
>>> M = [2, "turtle", 11, ["spam", "spamity", "spam"]]
>>> len(M)
4
>>> M[2]
11
>>> M[3]
['spam', 'spamity', 'spam']
```

```
>>> M[3][0]
???
>>> M[2:]
???
```

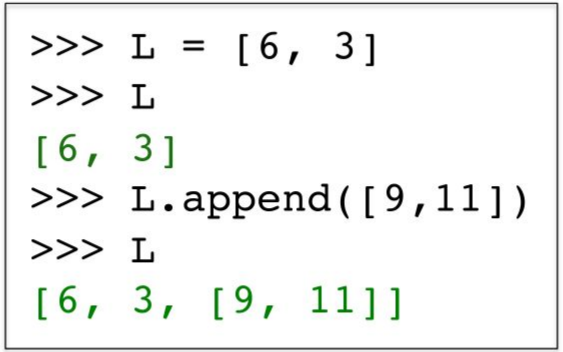
Addition and multiplication for lists

```
>>> my_list = [42, 47, 23]

>>> new_list = my_list + 100

TypeError: can only concatenate
list (not "int") to list

>>> new_list = my_list + [100]

>>> new_list

[42, 47, 23, 100]
```

```
>>> my_list

[42, 47, 23]

>>> new_list = my_list * 2

>>> new_list

[42, 47, 23, 42, 47, 23]
```

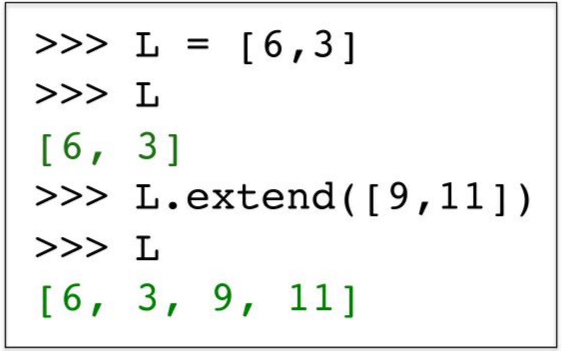
Special functions for adding elements to lists

`append`



```
>>> L = [6, 3]
>>> L
[6, 3]
>>> L.append([9,11])
>>> L
[6, 3, [9, 11]]
```

`extend`



```
>>> L = [6,3]
>>> L
[6, 3]
>>> L.extend([9,11])
>>> L
[6, 3, 9, 11]
```

Nothing is returned! L is **modified** instead!

Extend/Append Modify the Variable

```
>>> L = [6, 3]

>>> L

[6, 3]

>>> L + [9,11]

[6, 3, 9, 11]

>>> L

[6, 3]

>>> L.extend([9,11])

>>> L

[6, 3, 9, 11]
```

Operators like “+” **return** a new value but **don’t ASSIGN** it to the original variable.

```
>>> x = 5

>>> x + 3

8

>>> x

5
```

Where strings and lists differ: mutability.

```
>>> L = [29, 47, 17, 23]
>>> L
[29, 47, 17, 23]
>>> L[1] = 42 # change AKA mutate the list at index 1
>>> L
[29, 42, 17, 23] # lists are mutable
>>> S = "spam"
>>> S[1] = "c" # strings are immutable - you can’t change directly
TypeError: 'str' object does not support item assignment
>>> S = “scam” # need to assign a new string overwriting the variable
```

Where strings and lists differ: mutability.

```
>>> L = [29, 47, 17, 23]
>>> L.append(10)
>>> L
[29, 47, 17, 23, 10]
>>> S = "spam"
>>> S.append(“!”) # strings are immutable - you can’t append
AttributeError: 'str' object has no attribute 'append'
>>> S = S + “!” # need to assign a new string overwriting the variable
>>> S
“spam!”
```

Immutable lists: tuples

```
> (1, 2, 3)
(1, 2, 3)
> ()
()
> (42)
42
```

```
> (42,)
(42,)
> 1, 2
(1, 2)
> 42,
(42,)
```

```
> x = (3, 7)
> x
(3, 7)
> x = 4, 6
> x
(4, 6)
> x[1] = 42
TypeError: 'tuple' object does not
support item assignment
```

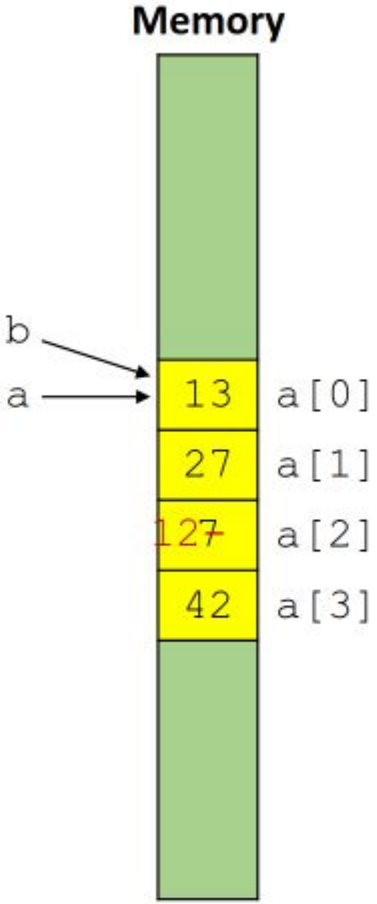
Eagle-eyed amongst you: I used these when explain any and all with booleans

Aliases: a common python gotcha

When compound + mutable:

b is assigned to **a** **NOT** the value of **a**

```
>>> a = [ 13, 27, 7, 42]
>>> b = a
>>> b
[ 13, 27, 7, 42]
>>> a[2] = 12
>>> b
[ 13, 27, 12, 42]
>>> b[2] = ‘a’
>>> a
[ 13, 27, ‘a’, 42]
```

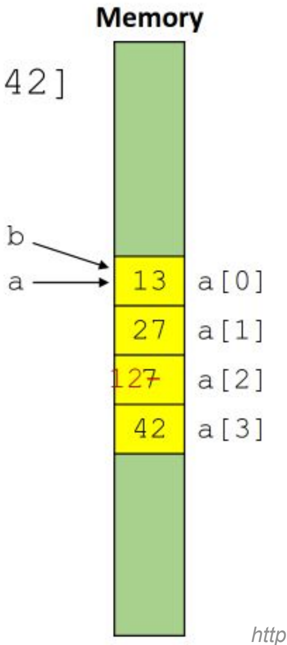


https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

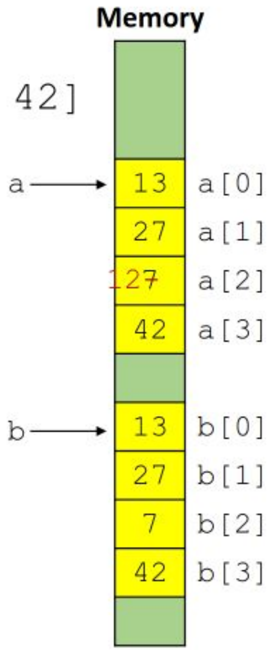
If you want y to be the value of x you need to **COPY**

`y = x` vs `y = x[:]`

```
a = [13, 27, 7, 42]
b = a
a[2] = 12
```



```
a = [13, 27, 7, 42]
b = a[:]
a[2] = 12
```



More complicated nested objects need **copy.deepcopy**

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Generating a list of numbers with “range”

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(2, 9, 2))
[2, 4, 6, 8]
```

- Same option start, stop, increment as list indexing.
- range(4) same as range(0, 4, 1)

Many list related functions do “lazy evaluation”

```
>>> x = enumerate([‘a’, ‘b’, ‘c’])

>>> x

<enumerate at 0x738dcf5156c0>


>>> list(x)

[(0, 'a'), (1, 'b'), (2, 'c')]
```

- Enumerate gives use the the index - value pairs

- Imagine x is very very big

- What if we only needed to enumerate the first couple of items in the list?

- Lazy evaluation means only doing calculations when (and therefore **IF**) they are actually needed

Zip efficiently combines lists

```
>>> x = [10, 50, 100]

>>> y = [‘a’, ‘b’, ‘c’]

>>> zipped = zip(x, y)

>>> zipped

<zip at 0x738dce007ec0>

>>> list(zipped)

[(10, 'a'), (50, 'b'), (100, 'c')]
```

How do I avoid writing lots of code to do something to every item in a list?

Loops - in python they can basically just be english!

```
list_of_numbers = [1,2,4]
for number in list_of_numbers:
    print(number + 1)


for character in 'abc':
    print(character + “!”)
```

- For every element in a sequence execute a body of code:

```
 for var in sequence:
     body
```

- Sequences can e.g. be lists, strings, ranges

Loops only go over top layer in nested lists by default

```
nested_list = [‘a’, ‘b’, [1, 2, 3]]

for item in nested_list:

    print(item)

    if type(item) == list:

        for x in item:

            print(x)
```

```
‘a’

‘b’

[1, 2, 3]

1
2
3
```

Loops can be nested just like lists and conditionals

Break and continue can be used to control loops

Break lets us escape from the loop

```
for x in [‘a’, ‘b’, ‘c’]:
    if x == ‘b’:
        break
    print(f”In-loop {x}”)
print(‘Done’)

‘In-loop a’
‘Done’
```

Continue goes to next iteration

```
for x in [1, 10, 30]:
    print(x)
    if x < 2:
        continue
    print(f”{x} + 1”)
```

```
1
10
10 + 1
30
30 + 1
```

Range is often used to generate indices for strings

```
for x in range(5, 15, 3):
    print(x)

5
8
11
14
```

```
a = ‘abc’

b = ‘123’

for i in range(len(a)):

    print(a[i] + b[i])

a1

b2

c3
```

We can also iterate over more than 1 list with zip

```
list1 = [‘a’, ‘b’, ‘c’]
list2 = [‘1’, ‘2’, ‘3’]
for a,b in zip(list1, list2):
    print(a + b)


a1
b2
c3
```

Overview

- Conditionals (if, elif, else) allow branching
- Functions let us define code once and then run it many times
- Strings are complicated by python makes life easier (including built-in functions)
- Strings can include variables with f-strings and special characters using escape sequences.
- Lists are a mutable ordered collection of data (tuples are immutable).
- Lists and strings have similar indexing/slicing but differ in mutability
- Aliasing vs copying is an easy way to make mistakes in python
- For loops let us do something for every item in a list or string