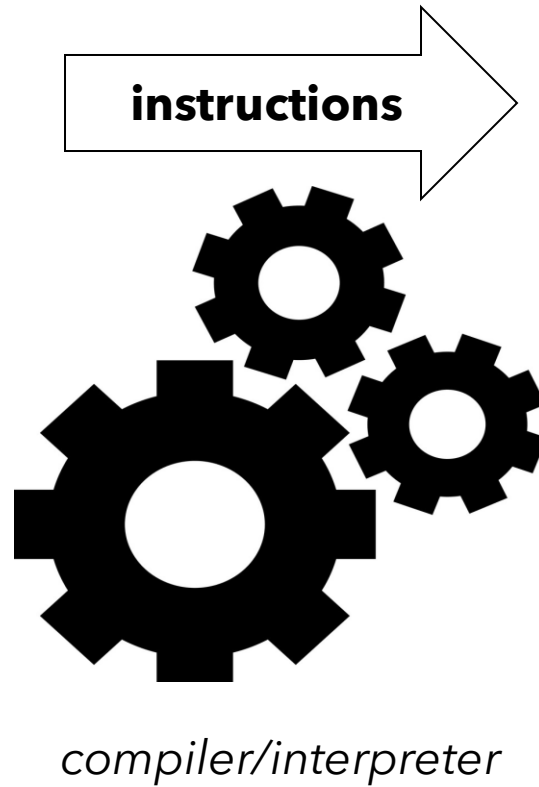


# A Crash Course on Getting Started

monica@utk.edu

# Programming Languages



# Installing Python – macOS Catalina

- Ships with a Python 2 and Python 3.
- Command Line / Terminal: *Go > Applications > Utilities > Terminal*
- Check version installed & associated with command:
  - `python --version`
  - `python3 --version`  
(example output: "Python 3.8.7")
- Check path to interpreter (installation location):
  - `which python`
  - `which python3`  
(example output: "user/bin/python3")

# Installing Python – Windows 10

- Confirm OS type: *Start > System > System type: 64 bit Operating system*
- On Python download page, choose "**Windows x86-64 executable installer**" version of download.
- Custom Install allows you to change the install location (recommended)
- Install for all users if account has admin privileges.
- Check yes for adding Python to PATH.
- Terminal access: From start menu search bar type cmd.exe
- Also check version with: `python3 --version`

# Installing Python – Windows 10

- Confirm OS type: *Start > System > System type: 64 bit Operating system*
- On Python download page, choose "**Windows x86-64 executable installer**" version of download.
- Custom Install allows you to change the install location (recommended)
- Install for all users if account has admin privileges.
- Check yes for adding Python to PATH.


# Installing Python – Windows 10

- At end of install: YES to disable path limit (allows greater than 260 character filepaths on windows).
- Terminal access: From start menu search bar type cmd.exe
- Also check version with: `python3 --version`

# Python Interactive Shell

- Simplest approach to executing Python code.
- From terminal, type `python3` to launch shell.
- Variables and computed values automatically displayed at **runtime**.

# Python Interactive Shell



The image shows a terminal window titled "monica@monica-Precision-7730: ~". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows the user running the command `python3` at the prompt `monica@monica-Precision-7730:~$`. The output displays the Python version and environment: `Python 3.6.8 (default, Oct 7 2019, 12:59:55)`, `[GCC 8.3.0] on linux`, and a message to type "help", "copyright", "credits", or "license" for more information. The prompt `>>>` is shown with a cursor.

```
monica@monica-Precision-7730: ~  
File Edit View Search Terminal Help  
monica@monica-Precision-7730:~$ python3  
Python 3.6.8 (default, Oct 7 2019, 12:59:55)  
[GCC 8.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> |
```



# From the Interactive Shell

```
print("hello world")
```

# Python Syntax - Commenting

- In line commenting: #
- Extends from the beginning of hash tag to the end of line.
- Comment generously: Your future self will thank you!

# Python Syntax – Variables

- Variables are locations in memory that store values.
- We identify variables by giving them names.
- Variables have data types: Numbers, strings, etc.
- These data types determine what you can do with a variable.

# Data Types



**Booleans** - can be True or False



**Numbers:**

*Integers* ( like 1 or 3)

*Floats* (such as 1.0 or 3.0)

*Fractions* (such as  $\frac{3}{4}$ )



**Strings** – Sequences of Unicode characters; Human readable.



**Bytes / byte arrays** – like a jpeg file - mutable sequence of integers in the range  $0 \leq x < 256$ . Not for human consumption!

# Python Syntax - Example Variables

- String: `my_string = "some text"`
- Float: `my_num = 3.1`
- Int: `also_my_num = 3`
- Boolean: `my_bool = False`
- List: `my_list = [1, 4, 9, 16, 25]`

# Python Syntax – White Space

- Leading whitespace / indentation (tabs, spaces) are used to indicate logical groupings of code.
- Spaces are recommended for compatibility reasons.
- New lines also contribute to logical groupings.
- Indentation is typically 2 or 4 spaces.
- Indication of a problem:  
`IndentationError: unexpected indent`

# Python Syntax: Flow Control

Flow control allows us to embed decision-making capabilities into our code:

```
x = 1
if x > 0:
    print("bananas")
```

# Syntax - Reserved Keywords

Cannot be used to identify variables, functions, classes, etc:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	



# Operators



**Assignment** – assigning values to variables



**Math** – Perform mathematical operations upon numbers.



**Comparison** – Return boolean values and can be chained.

Math Operator	Meaning	Example Output
+	addition	>>> 3 + 1 4
-	subtraction	>>> 10-4 6
*	multiplication	>>> 2*2 4
/	floating point division	>>> 10/4 2.5
//	integer division (be careful!)	>>> 10//4 # real answer is 2.5 2 >>> 10//4.0 # divide by a float 2.0
%	modulo (remainder after division)	>>> 10 % 3 1
**	exponent	>>> 2**3 8 >>> 2.00**3 # raising a float 8.0 >>>

Assignment Operator	Meaning	Example
=	assigns left to right	>>> a = 1
+=	left equals left plus right	>>> b = 1 >>> b += 3 4
-=	left equals left minus right	>>> c = 10 >>> c -= 3 7
*=	left equals left * right	>>> d = 2 >>> d *= 3 6
/=	left equals left divided by right	>>> e = 12 >>> e /= 3 4.0
**=	left equals left raised to the right power	>>> f = 3 >>> f **= 3 27
//+	integer division of left by the value of the right operator.	>>> g = 5 >>> g //= 2 2
%=	left = left mod right (remainder leftover after division)	>>> h = 10 >>> h %= 3 1

## + Operator vs Concatenation *(strings are joined, numbers are added)*

```
>>> team_data = "Kansas City: "  
>>> team_data = team_data + "31 points"  
>>> print(team_data)  
Kansas City: 31 points
```

```
>>> team_data = "Kansas City: "  
>>> team_data += "31 points"  
>>> print(team_data)  
Kansas City: 31 points
```

COMPARISON OPERATORS

==	Equal to	>>> 1 == 1 True >>> 1 == 2 False
!=	Not equal to	>>> 1 != 2 True
>	Greater than	>>> 1 > 2 False
<	Less than	>>> 1 < 2 True
>=	Greater than or equal to	>>> 10 >= 10 True >>> 10 >= 11 False
<=	Less than or equal to	>>> 3 <= 3 True
and	Boolean and	>>> 1 == 1 and 1 < 2 True >>> 1 <=3 and 0 > 1
or	Boolean or	>>> 1 > 10 or 1 < 10 True >>> 1 > 10 or 1 == 10 False
not	Boolean not	>>> not False True >>> not True False >>>

# Boolean Logic

or	Outcome
True or True	True
True or False	True
False or True	True
False or False	False

If either outcome evaluates to True, then the overall outcome is True

and	Outcome
True and True	True
True and False	False
False and True	False
False and False	False

Both outcomes must evaluate to True for the overall outcome to be True

# Boolean Logic

```
>>> sunny_day = True
>>> warm_temps = True
>>> if (sunny_day and warm_temps):
...     print('Go outside!')
...
Go outside!
```

# Boolean Logic

```
>>> dishes_done = True
>>> homework_done = False
>>> if (dishes_done and homework_done):
...     print('Time to relax')
...
>>> if (dishes_done and not homework_done):
...     print('Do your homework before you relax.')
...
Do your homework before you can relax.
>>>
```

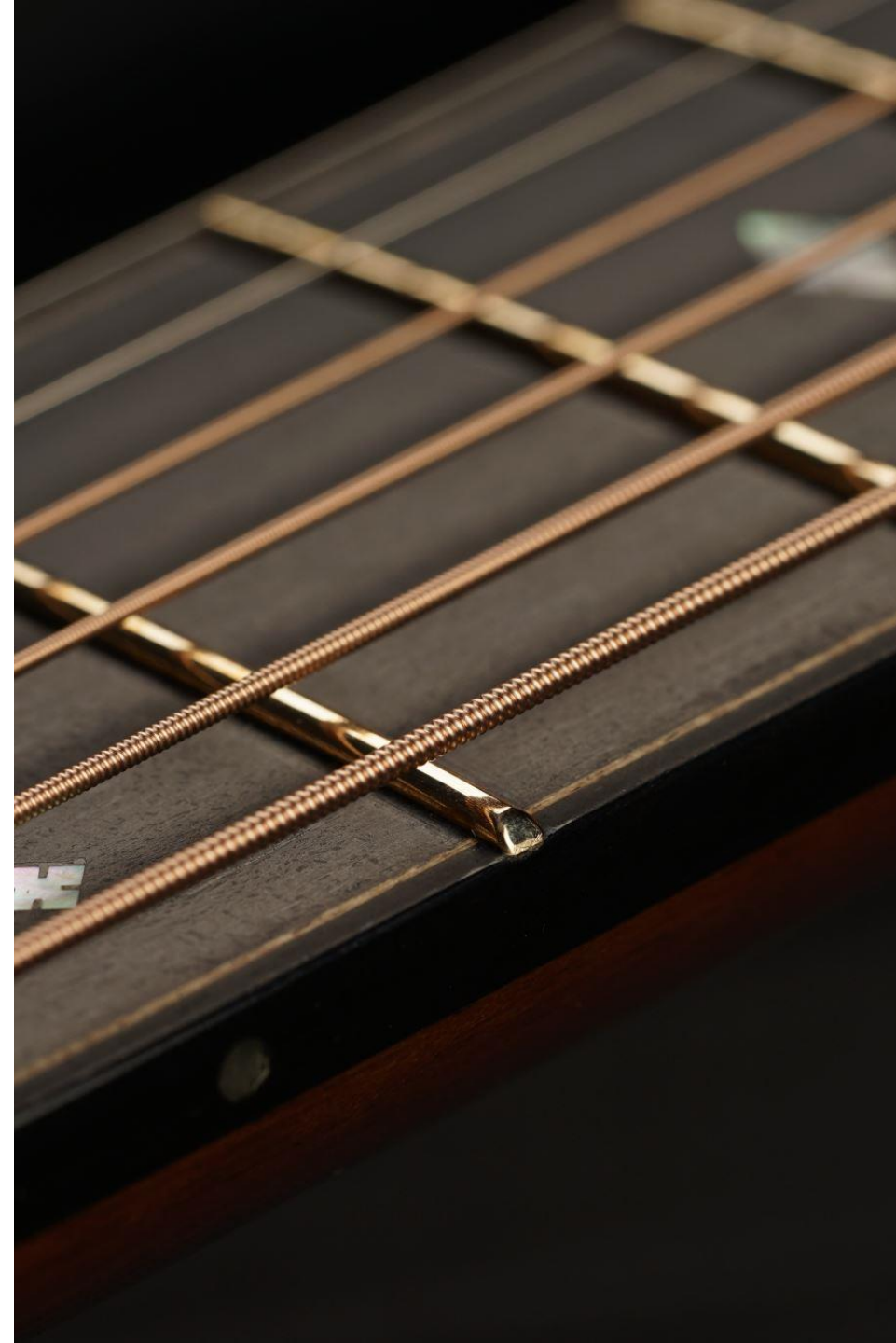


# Strings

---

- A **string** is simply a sequence of characters.
- Strings are **immutable**.
- Defined by enclosing text in quotes.
- Empty string can be declared with "".

```
my_string = "This is a test"  
my_empty_string = ""
```



# Escape Characters

Escape Sequence	Meaning
<code>\newline</code>	Backslash and newline ignored
<code>\\</code>	Backslash (\)
<code>\'</code>	Single quote (')
<code>\"</code>	Double quote (")
<code>\a</code>	ASCII Bell (BEL)
<code>\b</code>	ASCII Backspace (BS)
<code>\f</code>	ASCII Formfeed (FF)
<code>\n</code>	ASCII Linefeed (LF)
<code>\r</code>	ASCII Carriage Return (CR)
<code>\t</code>	ASCII Horizontal Tab (TAB)
<code>\v</code>	ASCII Vertical Tab (VT)
<code>\ooo</code>	Character with octal value ooo
<code>\xhh</code>	Character with hex value hh

# Python 3 Strings

- Can tell you things about the string:
  - `.islower()` - Is all lower case?
  - `.isnumeric()` - All numeric characters?
- Can alter the existing text:
  - `.upper()` - Convert to entirely uppercase
  - `.title()` - Convert to title case (all words start capitalized)
- Clean up messy text:
  - `.strip()` - Get rid of extra leading and trailing characters remove

# Python 3 Strings

---

- Data Types can be manipulated back and forth between string and numeric types:
  - `float()`
  - `int()`
  - `str()`



# Python 3 Strings

- Critical to text mining and processing!
- <https://docs.python.org/3/library/string.html>
- <https://docs.python.org/3/library/stdtypes.html>

# .format()

```
print(var1)
```

```
print( 'Here are some variables: ', var1, var2)
```

```
print(var1, var2)
```

```
print("Here are some variables: {}, {}".format(var1,var2))
```

# .format()

<https://docs.python.org/3/library/string.html#format-specification-mini-language>

```
num = 123
```

```
print("Format the number {0:f}".format(num)) # Output: Format the number 123.000000
```

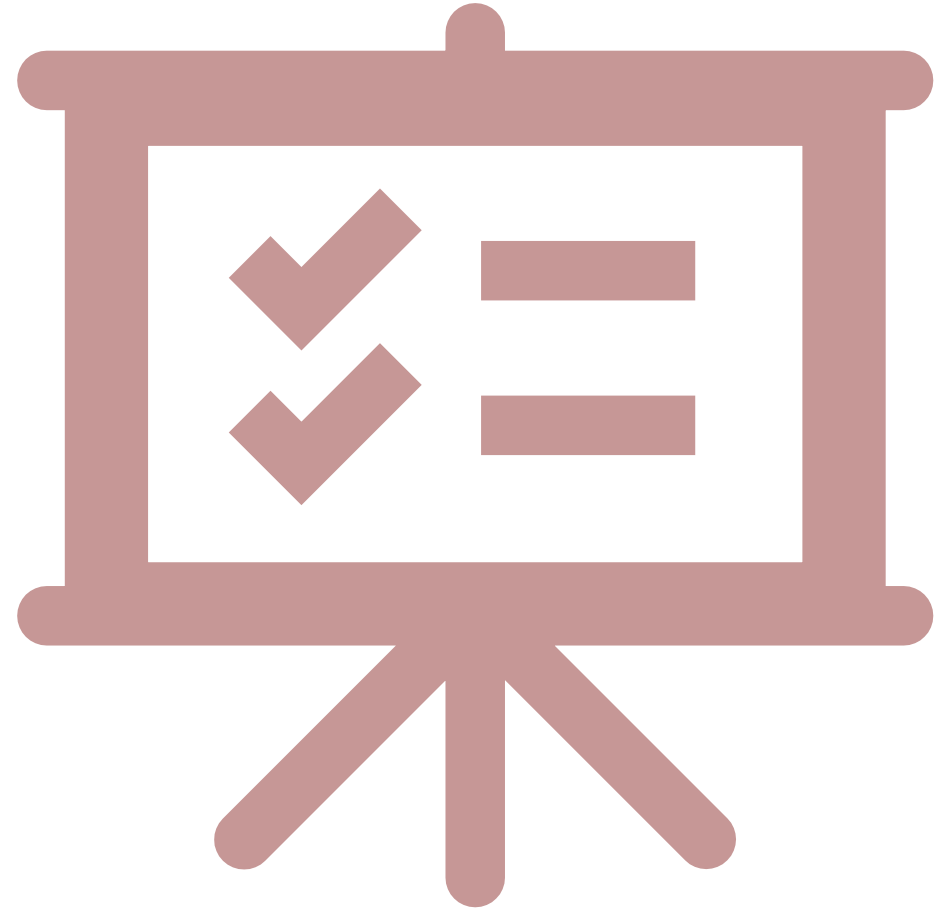
```
print("Format the number {0:.2f}".format(num)) # Output: Format the number 123.00
```

```
# scientific notation
```

```
print("Format the number {0:e}".format(num)) # 1.230000e+02
```

# Containers

- Containers are objects that hold other objects.
- Examples:
  - dict
  - list
  - set
  - tuple





# Lists

- Holds a sequence of items.
- Can hold multiple data types.
- Items are referenced by their index.
- Created using `[ ]`

```
my_list = ['bread', 'milk', 'eggs']  
print(my_list)
```

*Output:*     `['bread', 'milk', 'eggs']`

```
▼ my_list = {list} <class 'list': ['bread', 'milk', 'eggs']  
  01 0 = {str} 'bread'  
  01 1 = {str} 'milk'  
  01 2 = {str} 'eggs'  
  01 __len__ = {int} 3
```

# Working with Lists

## Count the Number of Items

```
my_list = ['bread', 'milk', 'eggs']  
count = len(my_list)  
print(count)
```

*Output:*    3

## Iterate over Contents with a for loop

```
my_list = ['bread', 'milk', 'eggs']  
for item in my_list:  
    print(item)
```

*Output:*

*bread*  
*milk*  
*eggs*

# Working with Lists

## Append New Item

(Add new item to end of list)

```
my_list = ['bread', 'milk', 'eggs']  
my_list.append('apples')  
print(my_list)
```

*Output:*

```
['bread', 'milk', 'eggs', 'apples']
```

## Extend A List

(Copy a sequence over to a list in place, no return)

```
list1 = [1,2,3]  
list2 = [4,5,6]  
list1.extend(list2)  
print(list1)  
print(list2)
```

*Output:*

```
[1, 2, 3, 4, 5, 6]  
[4, 5, 6]
```

# Working with Lists

## Insert New Item

(Add item at specified position in existing list)

```
list1 = [1,2,3]
list1.insert(1,'a')
print(list1)
```

*Output:* [1, 'a', 2, 3]

```
list1.insert(len(list1),'b')
print(list1)
```

*Output:* [1, 'a', 2, 3, 'b']

## Remove Specific Values

(Raises error if no such value found )

```
my_list = ['bread', 'milk', 'eggs']
my_list.remove('milk')
print(my_list)
```

*Output:* ['bread', 'eggs']

# Working with Lists

## Count Occurrences of a Value

```
my_list = ['no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']  
  
print("{} people voted yes, and {} people voted no."  
      .format(my_list.count('yes'), my_list.count('no')))
```

*Output:*

*4 people voted yes, and 5 people voted no.*

## Sort Items Based on Value

```
list1 = ['d', 'g', 'c', 'a', 'b', 'f', 'e']  
list1.sort() # modifies in place!  
print(list1)  
list1.sort(reverse=True)  
print(list1)
```

*Output: ['a', 'b', 'c', 'd', 'e', 'f', 'g']*

# Slicing Lists

- Slicing uses format: `list_name[ start : end]`
- Start means an integer index of position to start the slice, including that index.
- End is where to stop the slice, **NOT** including that end index.

# Slicing Lists

a	b	c	d	e	f	g	h	i
0	1	2	3	4	5	6	7	8

```
start = 0
end = 3
list1 = ['a','b','c','d','e','f','g','h','i']
print(list1[start:end])
print(list)
```

*Output:*

`['a', 'b', 'c']`

`['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']`

a	b	c	d	e	f	g	h	i
0	1	2	3	4	5	6	7	8

```
list1 = ['a','b','c','d','e','f','g','h','i']
print(list1[2:6])
```

*Output:*

`['c', 'd', 'e', 'f']`

# Slicing Lists

a	b	c	d	e	f	g	h	i
0	1	2	3	4	5	6	7	8

```
list1 = ['a','b','c','d','e','f','g','h','i']  
print(list1[1:7])
```

*Output:*  
['b', 'c', 'd', 'e', 'f', 'g']

milk	eggs	bread	cake	coffee	rice
0	1	2	3	4	5

```
shopping_list = ['milk', 'eggs', 'bread',  
                 'cake', 'coffee', 'rice']  
print(shopping_list[2:5])
```

*Output:*  
['bread', 'cake', 'coffee']



# Removing Elements with Slicing

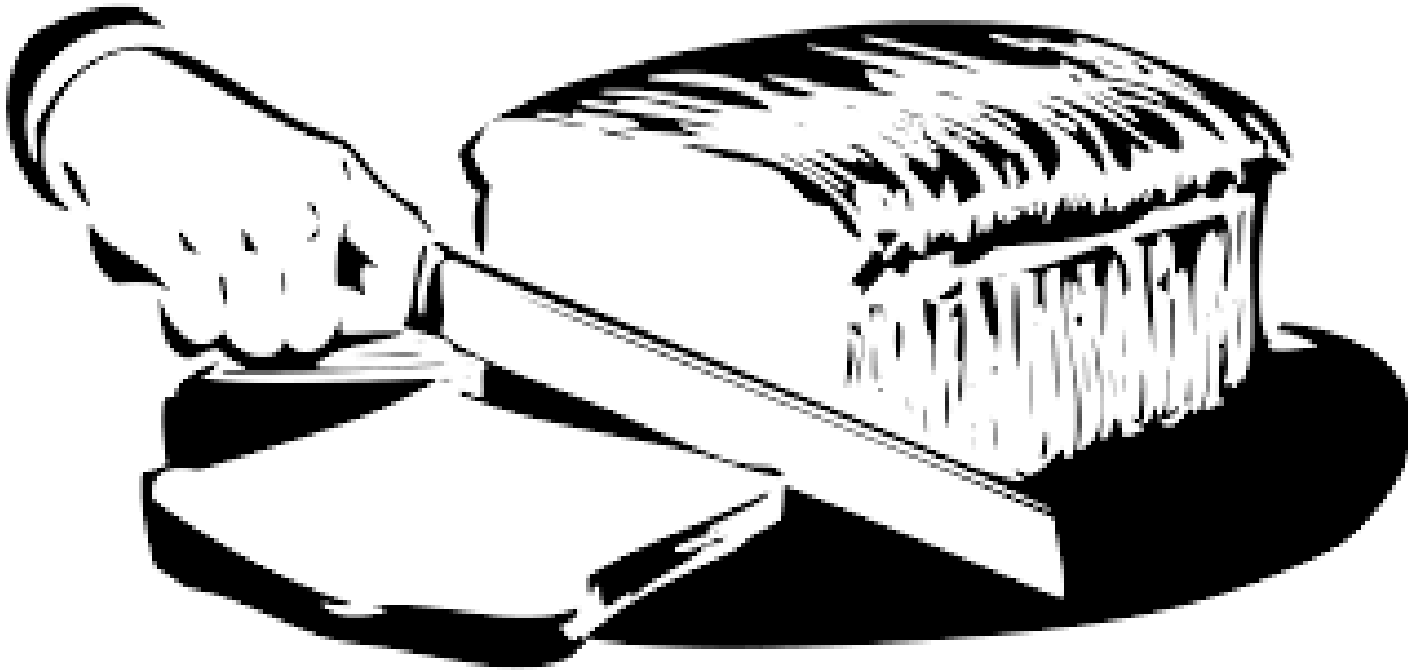
```
shopping_list = ['milk', 'eggs', 'bread', 'cheesecake', 'coffee', 'rice']
```

```
# deleting a slice  
del shopping_list[2:5]  
print(shopping_list)
```

*Output:*  
['milk', 'eggs', 'rice']



# Slicing Works for Strings too!



```
my_string =  
    "Time for a coffee break."  
print(my_string[11:17])
```

*Output: coffee*

# list of lists (Yes, that's a thing)

```
list_of_lists = []
```

```
list1 = ['a','b','c']
```

```
list2 = ['d','e','f']
```

```
list3 = ['g','h','i']
```

```
list_of_lists.append(list1)
```

```
list_of_lists.append(list2)
```

```
list_of_lists.append(list3)
```

```
print(list_of_lists[0][2]) # prints c
```

```
print(list_of_lists[1][1]) # prints e
```

```
print(list_of_lists[0][0]) # prints a
```

```
list_of_lists = {list} <class 'list': [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
0 = {list} <class 'list': ['a', 'b', 'c']
1 = {list} <class 'list': ['d', 'e', 'f']
2 = {list} <class 'list': ['g', 'h', 'i']
__len__ = {int} 3
```

# list of lists

```
list_of_lists = []
```

```
list1 = ['a', 'b', 'c']
```

```
list2 = ['d', 'e', 'f']
```

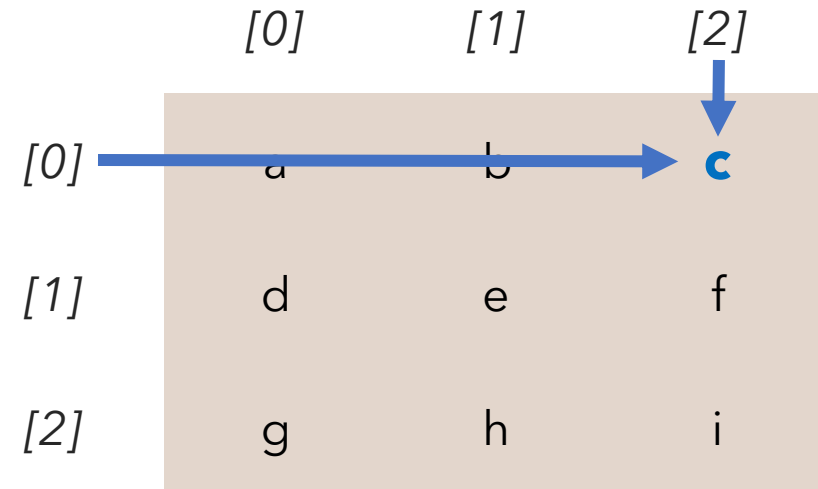
```
list3 = ['g', 'h', 'i']
```

```
list_of_lists.append(list1)
```

```
list_of_lists.append(list2)
```

```
list_of_lists.append(list3)
```

```
print(list_of_lists[0][2]) # prints c
```



# list of lists

```
list_of_lists = []
```

```
list1 = ['a', 'b', 'c']
```

```
list2 = ['d', 'e', 'f']
```

```
list3 = ['g', 'h', 'i']
```

```
list_of_lists.append(list1)
```

```
list_of_lists.append(list2)
```

```
list_of_lists.append(list3)
```

```
print(list_of_lists[0][2]) # prints c
```

```
print(list_of_lists[1][1]) # prints e
```

```
print(list_of_lists[0][0]) # prints a
```

	[0]	[1]	[2]
[0]	a	b	c
[1]	d	e	f
[2]	g	h	i

# list of lists

## Trying to Access Index that Doesn't Exist (This is true for any list):

```
File "/home/monica/PycharmProjects/insc360/Feb-10/lists.py", line 116, in <module>  
    print(list_of_lists[0][3])  
IndexError: list index out of range
```

	[0]	[1]	[2]
[0]	a	b	c
[1]	d	e	f
[2]	g	h	i

# dictionary

- Indexed by keys instead of position.
- Values can be any type but keys must be immutable (e.g. number or string).
- Basically a set of key-value pairs
- A few syntax options for creating dictionaries (also short-version is "dict")

# dictionary

```
staff_dict = { 'emp_id_code' : 'mktg1234', 'name' : "Samantha", 'years': 3 }
```

```
▼ ■ staff_dict = {dict} <class 'dict'>: {'emp_id_code': 'mktg1234', 'name': 'Samantha', 'years': 3}  
  01 'emp_id_code' (140569265994800) = {str} 'mktg1234'  
  01 'name' (140569342213008) = {str} 'Samantha'  
  01 'years' (140569266004744) = {int} 3  
  01 __len__ = {int} 3
```



# dictionary

## THREE WAYS TO DO THE SAME THING

***Probably most common way:***

```
staff_dict = { 'id_code' : 'mktg1234', 'name' : "Samantha", 'years': 3 }
```

***Least amount of typing:***

```
staff_dict = dict(id_code='mktg1234', name='Samantha', years=3)
```

***Yet another way:***

```
staff_dict = dict([('id_code', 'mktg1234'), ('name', 'Samantha'), ('years', 3)])
```

# dictionary

```
staff_dict = { 'id_code' : 'mktg1234', 'name' : "Samantha", 'years': 3 }
```

```
print(staff_dict['id_code']) # prints the value associated with the key that you just gave it  
print(staff_dict['name'])  
print(staff_dict['years'])
```

*Output:*

```
mktg1234  
Samantha  
3
```

# dictionary

```
staff_dict = { 'id_code' : 'mktg1234', 'name' : "Samantha", 'years': 3 }
```

```
print(staff_dict['not_existing']) # try to access a key that doesn't exist
```

Traceback (most recent call last):

File "/home/monica/PycharmProjects/insc360/Feb-10/dictionary.py", line 20, in  
<module>

```
{'id_code': 'mktg1234', 'name': 'Samantha', 'years': 3}
```

```
print(staff_dict['not_existing'])
```

**KeyError: 'not\_existing'**

# Removing Data with del

**del** staff\_dict['id\_code'] *# remove entry*

staff\_dict.clear() *# remove all entries, leaving an empty dictionary*

**del** staff\_dict *# delete the entire dictionary*

# List of Dictionaries

```
my_cats = [  
    dict(name='Salsa', age='7', color='black'),  
    dict(name='Zippy', age='6', color='black'),  
    dict(name='Velvet', age='7', color='gray')  
]
```

```
▼ my_cats = {list} <class 'list'>: [{'name': 'Salsa', 'age': '7', 'color': 'black'}, {'name': 'Zippy', 'age': '6', 'color': 'black'}, {'name': 'Velvet', 'age': '7', 'color': 'gray'}]  
▶ 0 = {dict} <class 'dict'>: {'name': 'Salsa', 'age': '7', 'color': 'black'}  
▶ 1 = {dict} <class 'dict'>: {'name': 'Zippy', 'age': '6', 'color': 'black'}  
▶ 2 = {dict} <class 'dict'>: {'name': 'Velvet', 'age': '7', 'color': 'gray'}  
01 __len__ = {int} 3
```

# List of Dictionaries

```
my_cats = [  
    dict(name='Salsa', age='7', color='black'),  
    dict(name='Zippy', age='6', color='black'),  
    dict(name='Velvet', age='7', color='gray')  
]
```

```
for cat in my_cats:  
    print('I have a cat named {} who is {} years old.'.format(cat['name'], cat['age']))
```

Output:

*I have a cat named Salsa who is 7 years old.  
I have a cat named Zippy who is 6 years old.  
I have a cat named Velvet who is 7 years old.*

# List of Dictionaries

```
my_cats = [  
    dict(name='Salsa', age='7', color='black'),  
    dict(name='Zippy', age='6', color='black'),  
    dict(name='Velvet', age='7', color='gray')  
]  
  
print(my_cats[0]['name'])  
print(my_cats[1]['name'])  
print(my_cats[2]['name'])
```

*Output:*

*Salsa*

*Zippy*

*Velvet*

# Tuples

---

Sequences of values

---

Immutable (unlike lists which are mutable)

---

Must contain heterogeneous items (no mixing up data type)



# List vs Tuple

---

List:	Use lists if your data will change frequently, for performance efficiency.
List:	Use list for more features like sorting.
Tuple:	Use tuple to conserve memory and for situations where data will not change.
Tuple:	Tuples will give you better processing performance for accessing values quickly, for fixed size data that won't change.

# Tuples

my\_tuple = ('a', 'b', 'c', 'd', 'e')

```
1 my_tuple = {tuple} <class 'tuple'>: ('a', 'b', 'c', 'd', 'e')
01 0 = {str} 'a'
01 1 = {str} 'b'
01 2 = {str} 'c'
01 3 = {str} 'd'
01 4 = {str} 'e'
01 __len__ = {int} 5
```

my\_other\_tuple = (1,2,3,4,5)

```
1 my_other_tuple = {tuple} <class 'tuple'>: (1, 2, 3, 4, 5)
01 0 = {int} 1
01 1 = {int} 2
01 2 = {int} 3
01 3 = {int} 4
01 4 = {int} 5
01 __len__ = {int} 5
```

# Tuples

```
print('Iteration:')  
for item in my_tuple:  
    print(item)
```

```
print('By index:')  
print(my_tuple[0])  
print(my_tuple[1])  
print(my_tuple[2])  
print(my_tuple[3])  
print(my_tuple[4])
```

*Output for both:*

*a  
b  
c  
d  
e*

# Sets



Unordered and unindexed collections (meaning you can't access by a positional index the way you can with lists and dictionaries).



No duplicates



Like other collections, sets support **`x in set`**, **`len(set)`**, and **`for x in set`**.

# Creating Sets

```
set1 = {'Knox', 'Anderson', 'Roane', 'Sevier'}
```

```
set2 = {'Anderson', 'Knox', 'Anderson', 'Knox', 'Knox', 'Roane',  
       'Sevier', 'Anderson'}
```

```
print(set1)
```

```
print(set2)
```

*Output for both: {'Sevier', 'Anderson', 'Roane', 'Knox'}*

```
▼ set1 = {set} <class 'set': {'Sevier', 'Anderson', 'Roane', 'Knox'}  
01 140248355219024 = {str} 'Sevier'  
01 140248355174064 = {str} 'Anderson'  
01 140248355218576 = {str} 'Roane'  
01 140248355218688 = {str} 'Knox'  
01 __len__ = {int} 4
```

# Creating & Adding to Sets

```
# set from a list  
my_list = ['cookies', 'cookies', 'cake', 'pie', 'cookies', 'cake']  
set3 = set(my_list)  
print(set3)
```

*Output: {'pie', 'cake', 'cookies'}*

```
# adding to a set  
set3.add('churros')  
print(set3)
```

*Output: {'pie', 'cake', 'churros', 'cookies'}*