Python Commands
With Commentary

(for learning and homework; NOT for exams or quizzes)

Hello, World!

```python
"""
A classic hello-world program.

:author: T.Sergeant
"""

print("Hello, World!") # we greet the world with optimism!
```

Expected output (on screen):

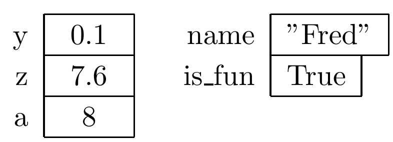
```
Hello, World!
```

It is a time-honored tradition in computer science to begin any new endeavor by creating the simplest possible working example. When writing a program that often involves getting the computer to display something. So, this is a Python “Hello, World” program. The idea is that if you can get this program to compile and run on your computer then you have all the pieces you need to begin writing more complex progams. Some things to note:

- Python is famed for its concise notation. We don’t need any “setup” commands or other structures. We just enter the commands we want to happen.
- Comments are used to provide notes/annotation to be read by other programmers. Comments are ignored by the compiler. To create a multi-line comment use `"""` at the beginning and end of the comment block. We call such a comment a “docstring”. A single-line comment starts with `#` and goes to the end of the line.

Variables and Simple Types

```python
y = 0.1
z = 7.5 + y
name = "Fred" # quotes indicate a string
a = int(z)    # treat z as an int
a = a + 1
is_fun = True
```



We don’t “declare” variables in Python as we do in some other languages. We just start using variables when we need them. Variables do have a “type” associated with them but their type can change depending on the value currently stored in the variable. Some simple variable types supported by Python:

float	numbers with decimal places
int	integers (i.e., numbers without decimal places)
str	strings; zero or more characters (like words)
bool	boolean; either `True` or `False`

Initializing a variable means “giving it a value for the first time.” Initializing a variable is accomplish through use of the *assignment operator* (i.e., =). The assignment operator takes a value (on the right) and places it into the variable (on the left). The statement `a = int(z) x;` is an example of type-casting (temporarily changing the type of an expression). When we *type-cast* a `float` to be an `int` the result is that the decimal places get chopped off. The statement `a= a + 1` highlights the different between the assignment operator and the mathematical concept of “equals”. In this case the variable `a` is incremented by 1.

Displaying Values

```python
a = 23
name = "Mary"
print("Hello")
print("Name is: ", end='')
print(name)
print("Answer is: " + str(a))
```

Output of statements assuming variables have values from previous examples:

```
Hello
Name is: Mary
Answer is: 23
```

The `print` command is used to display results to the console/screen. You simply put the value or constant you want to print in parentheses. After displaying the value `print` performs an extra step of displaying a newline character (so future output will appear on the following line). To print without going to the next line you can add the `end=''` inside the print to indicate no newline should be used. This is why "Name is: " and "Mary" appear on the same line.

You can also combine multiple strings by using "+" to concatenate (i.e., stick strings together) them before printing. In this example the variable `a` as an `int` so we had to typecast it to a string before doing the concatenation.

Getting Input

```python
word = input("Enter word: ")
age = input("Enter age: ")
next = int(age) + 1
num = float(input("Enter a number: "))
num = num + 2
```

```
Enter word: fun
Enter age: 10
Enter a number: 7.5
```

word	fun
age	10
next	11
num	9.5

The `input` command does several tasks: (1) it displays a message to the screen, (2) it pauses and waits for the user to type something, (3) when the user types something followed by the `<Enter>` key it returns what they typed as a string value.

Even if the user types a number the result of the `input` command is always a string. So, if you want to do calculations with the value you have to type-cast the result. This can be done afterwards in the calculation (e.g., as with `age`) or can be done immediately (e.g., as with `number`).

Integer Arithmetic and Random Numbers

```python
import random
random.seed()  # do this once per program

a = 7
b = -1
c = a + b * 3 # 4
d = b - a // 3 # -3
e = b - a / 3 # -3.33333
f = a % 3     # 1
g = random.randint(1,100) # random val in range 1 to 100
```

a	7
b	-1
c	4
d	-3
e	-3.333333
f	1
g	57

Integer arithmetic in Python is fairly simple. You can use *arithmetic operators* ($-$, $+$, $*$, $/$) to perform addition, subtraction, multiplication, and division, respectively. The order of operations works as in algebra. Parentheses can be used to override order of operations. NOTE: Dividing integers has a wrinkle in that Python actually provides three commands for accomplishing division:

/	"normal" division that results in a float (in Python 3); see calculation of `e`
//	integer division that results in an int; $a//3$ results in 2 (because 3 goes into 7 two times with a remainer of 1)
%	integer division that produces the remainder (or modulus)

The `random.randint(x,y)` function will provide a randome value between `x` and `y` (inclusive). It can be called multiple times to get different random values. The `random.seed()` function should be called only once when the program starts.

Floating Point Arithmetic

```
import math
a = 4.0
b = -2.2
c = a + b * 3          # -2.6
d = math.sqrt(a) + b # -0.2
```

a	4.0
b	-2.2
c	-2.6
d	-0.2

Floating point arithmetic works like integer arithmetic except that division does not produce an integer. In addition there are a host of functions provided in the `math` library to provide many useful results. This example shows how to find the square root of a number. For a complete list of `math` functions see: `https://docs.python.org/3/library/math.html`.

If Statements

```
if a < 0:
   print("a is negative")
   a = 0
else:
   print("a is non-negative")

if b == 1:
    print("hi")
elif b == 2:
   print("there")
elif b == 3:
   print("how")
else:
   print("are")
```

The *if-statement* allows you to select one of two alternate actions based on a comparison. After the word "if" you must have a boolean expression (which evaluates to either true of false) and then a colon. If the expression evaluates to true the statements that are indented below the `if` in the first section will happen. Python uses indentation to define the scope of various operation. In the case the boolean expression evaluates to false, the `else` part will happen. Keep in mind the statements in either section could be if-statements themselves (which is called "nested ifs"). The `else` portion of the statement is optional. If else is omitted and the boolean expression is false then the statements are simply skipped.
The second example above shows the common case where we may want to put an if-statement inside an else block. Python provides the shorthand `elif` to handle this case.

Boolean Expressions

```python
is_cool = False
a = 7
b = -1
c = True
d = a < b
e = a < b or a > 0
f = a <= b and a != 0
g = not is_cool and a == 7
h = 0 <= b <= 100
```

a	7
b	-1
c	True
d	False
e	True
f	False
g	True
h	False

A *boolean expression* is an expression that evaluates to either `True` or `False`. NOTE: Python requires those two constants to be written with an initial capital letter and no quotes. Boolean expressions can be built using a combination of variables, relational operators, boolean operators, and constants. *Relational operators* (<, <=, >, >=, ==, ! =) compare values of the same type and produce a boolean result. NOTE: == compares values for equality and is different than = (which is the assignment operator). Also, ! = means "not equal". *Boolean operators* (`and`, `or`, `not`) act on boolean values and produce a boolean result. NOTE: `and` is true only if both parts are true. The boolean operator `or` is false only if both parts are false. When placed in front of a boolean expression the operator `not` toggles the expression's value. Parentheses can be used to specify the desired order of evaluation of a complex boolean expression.

Python is a bit unusual in that it honors the notation specified for the variable `h` in the example above.

Strings

```python
one = "Fun"
two = 'ny'
num = len(one)
three = one + two
print(three[1])
print(three[1:4])
if one < two:
  print("Fun is less than ny")
print(three.upper())
print(three.lower())
four = "   " + three + " guy"
print(four.strip())  # remove whitespace from beg and end
```

num	3
one	"Fun"
two	"ny"
three	"Funny"

Expected output:

```
u
unn
Fun is less than ny
FUNNY
funny
Funny guy
```

Strings are used to hold a series of characters. String constants can be specify using either single or double quotes. The `len` function returns the length of a string. When the `+` operator is applied to strings it performs *string concatenation*, which is the fancy computer science terms that means "stick them together". To access an individual character in a string you can specify the index in `[]`'s with 0 being the index of the first character. You can also apply Python's slice notation to strings to extract a substring. The `upper()` and `lower()` functions return an upper- and lower-case copy of the string, respectively.

When comparing strings to one another, Python performs a lexicographic comparison of the individual characters of the strings moving from the first character to the last, stopping when it finds non-matching characters. So, if comparing "act" and "apple" it will look at the first letter in each word. Since those match it will move to the second letter. Since those are different it will compare those two letters to determine which is "bigger" than the other. Whether one letter is bigger than another is based on the ASCII chart (`http://www.asciitable.com/`. So, python would say that "act" is less than "apple" and that "act" is greater than "APPLE" (because lower and upper case letters have different ASCII values).

While Loops

```python
a = 1
while a <= 8:
    print(a)
    a += 2   # shorthand for a = a + 2
```

Expected output:

```
1
3
5
7
```

A *while loop* is used to repeat a group of actions and begins with the keyword `while` followed by a boolean expression followed by a colon. The Python statements indented below the `while` are run if the boolean expression evaluates to `True`. When the statements have been executed the boolean expression is revaluated to determine whether or not the statements should be performed again. If the expression evaluates to `False` the statements are skipped. For this reason the boolean expression must contain a variable that gets changed by the statements in the loop (or else suffer the fate of a so-called "infinite loop").
Also, note the use of the `+=` operator which is simply shorthand for adding a value to a variable.

Lists

```python
a = ["a", 12, "fun", "hi", True, 17.3]
print(len(a))
print(a[3])
#a.sort()
print(a)

b = []           # b is an empty list
b = b + ["fun"] # stick two lists together
b.append("neat") # add to end of list
b.insert(0, "two") # add to front of list
print(b)

c = a[1:4]
print(c)

val = a.pop()   # remove and return last element
a.remove("fun") # remove the value "fun" from list
del a[0]        # remove value a position 0
print(a)
print(val)

num = 12
if num in a:
   print("Found it!")
else:
   print("Not found.")
```

Expected output:

```
6
hi
[’a’, 12, ’fun’, ’hi’, True, 17.3]
[’two’, ’fun’, ’neat’]
[12, ’fun’, ’hi’]
[12, ’hi’, True]
17.3
Found it!
```

Lists in Python are delineated using `[]`'s and can hold values of various types. Each value can be extracted by specifying its index (starting with 0). The `len()` function can be used to determine the number of elements in a list. The `sort()` function re-arranges the list to its natural sorted order. There are multiple ways of adding elements to a list and of removing elements from a list as illustrated above. To search for a value in a list use the `in` operator. A section of a list can be extracted using Python's "slice" notation: `a[1:4]` returns a section of `a` starting with the index 1 and going up to (but not including) the value indexed by 4.

For Loops

```python
a = ["hi", 12, "fun"]

for val in a:
   print(val)

for i,val in enumerate(a):
   print(str(i)+" "+str(val))

for num in range(4):
   print(num)
```

Expected output:

```
hi
12
fun
0 hi
1 12
2 fun
0
1
2
3
```

A *for loop* is user to iterate through a list or other structure in Python. The first loop in this example iterates through the list of 3 elements and `val` takes on the value of each of those elements. The second loop is similar but the use of the `enumerate()` function provides both the index of each element and its value. The `range()` function is used to produce a loop that happens a fixed number of times (4 times in this case) and the variable `num` takes on integer values starting at 0.

Functions

```python
def show_sum(a, b):
   c = a + b
   print("Sum is: "+str(c))

def calc_sum(a, b):
   c = a + b
   return c

# To call these functions ...
x = 7.5
show_sum(x, 1.0)
show_sum(x * 2.0, x - 2.0)       # positional arguments
show_sum(b = x - 2.0, a = x * 2.0) # keyword arguments
x = calc_sum(1.0, 2.2)
print(x)
```

Expected output:

```
Sum is: 8.5
Sum is: 20.5
Sum is: 20.5
3.2
```

Writing a function allows you to give a name to a section of code. The section of code can then be invoked at any time by simply using the name of the function. It is useful to name sections of code to organize a large program or for sections that will need to be invoked multiple times. To define a function start with the keyword `def` followed by the name of the function followed by the (formal) parameter list. Names of functions should be verbs that describe the action of the function. The *parameter list* declares variables that must be filled in order for the function to do its job. In this example the functions specify two parameters named `a` and `b`. The functions must be *called* in order for them to run. This example shows three calls to `show_sum` and one call to `calc_sum`. When calling a function you must specify the *arguments* (or *actual parameters*) that will be used to fill in the formal parameters for that particular call. So, in the first call the parameter `a` takes on the value 7.5 and the parameter `b` takes on the value 1.0. This is an example of *positional arguments* because the values taken left-to-right are plugged in to the parameters left-to-right.

Arguments can also be passed using keyword notation as in the third call to `show_sum` where we explicitly assign values to parameters by name. In this situation naming the parameters overrides the natural left-to-right assignment.

The `calc_sum` function demonstrates a function that returns a value. In function that returns a value you must 1) have a return statement to indicate the value to the returned, and 2) do something with the returned value. In the example, the local variable `c` is returned and the returned value is stored into the variable `x`.

DocString / reST Notation

```python
def calc_sum(a, b):
    """
    Calculate and return the sum of two numbers.

    :param a: first number to be used in sum
    :param a: second number to be used in sum
    :return: sum of a and b
    """
    c = a + b
    return c
```

One convention to documenting functions is to use the so-called reST notation which has these characteristics:

- begins with three quotes and is indented under the function header
- first line is a single sentence ending with a period followed by a blank line
- each parameter is described using ":param nameOfParameter: description of parameter"
- if function returns a value then do ":return: description of what is returned"

Formatted Output

```python
x = 7.5
str = "Fred"
a = 2
print("%s is %d" % (str,a))
print("A%5d %4.2fB" % (a,x))
print("A%-5d %4.2fB" % (a,x))
```

Expected output:

```
Fred is 2
A    2 7.50B
A2     7.50B
```

It can be useful to to produce formatted output using printf-style notation because that notation is used widely across many languages and tools. Inside the print statement provide a format string followed by a percent sign followed by a list of values (in parentheses) you want to plug into the placeholders Placeholder can be: `%s` for strings, `%d` for ints, and `%f` for doubles/floats. The remaining parameters are values that will be plugged in to the placeholders. A number number right after the `%` in a format string is called a *field width specifier* and is used to specify how much space to leave for the value. In this example, `%4.2f` leaves 4 spaces for a floating point number that will display 2 decimal places. Placeholders with a field width specifier right justify the value by default. To left justify a value put a negative sign in front of the number.

Read Data File

```
with open("fun.txt") as first:
    for line in first:
       line = line.rstrip() # remove \n from end of line
       print(line)

with open("fun.txt") as again:
  list_of_lines = again.read().rstrip().split("\n")
print(list_of_lines)

third = open("fun.txt")
for line in third:
  print(line.rstrip())
third.close()
```

Expected output of first and third examples is contents of the file `fun.txt` displayed to the screen. The second example will display each line of the file as values in a Python list.

Working with data files requires three steps (just like using your refridgerator): open, use, close. The `with open` command handles the first and last steps it opens the file and used the variable you specify (`f` in first example) as the way to refer to the file in your program. When the `with open` block finishes it automatically closes the file. The `for` loop cycles through the file one line at a time (the "middle" step: use). We use `.rstrip()` to remove the newline character at the end of the line.

In the second example we read the entire file in a single statement (`.read()`) and then split it by line to create a Python list containing the lines of the file. The third example produces the same output as the first but handles the open/close without use of the `with` block.

Write to Data File

```
with open("cool.txt", "w") as my_file:
  my_file.write("Hello\n")
  my_file.write("Goodbye\n")

with open("cool.txt", "a") as second:
  for i in range(3):
    second.write(str(i)+"\n")
```

Expected output (in file "cool.txt"):

```
Hello
Goodbye
0
1
2
```

In the first example we open the file in "write mode" which causes the previous contents of the file to be completely overwritten. NOTE: The `.write()` command does not automatically append a newline at the end the way that `print()` does. So, we need to explicity add the newline if we want following writes to be on subsequent lines.

In the second example we open the file in "append mode" so the previous contents of the file are kept and values written are added to the end of the existing contents.

Timing Code

```python
import time
start = time.time()
# put code to be timed here
stop = time.time()
time_in_seconds = stop - start
print(time_in_seconds)
```

Expected output: the time elapsed in seconds

To time a section of code we import the `time` library. Each call to the `time.time()` function returns a timestamp. So we call the function before and after the code we want to measure and subtract the values to obtain time elapsed.

Dictionaries

```python
letter_counts = { "a": 67, "b": 12, "c": 16 }
print(letter_counts.keys())
print(letter_counts.values())
print(letter_counts["c"])
letter_counts["d"] = 9
del letter_counts["a"]
val = letter_counts.pop("b")
print(val)
if "b" in letter_counts:
    print("b is there")
else:
    print("b is gone")
for key, value in letter_counts.items():
    print(key+" has value "+str(value))
```

Expected output:

```
['a', 'b', 'c']
[67, 12, 16]
16
12
b is gone
c has value 16
d has value 9
```

A Python dictionary contains key/value pairs. In this example they keys are letters and the values are numbers. The function `.keys()` returns a list of keys in the dictionary. The function `.values()` returns a list of values in the dictionary. The function `.items()` returns both keys and values as tuples which we can iterate through. To access a particular value we index the dictionary with its key. We can use this same notation to add values to the dictionary (as demonstrated by adding ("d", 9)).
Removing elements from a dictionary by its key can be done using the `del` operator or by using the `.pop()` function. The latter returns the value being removed. The `in` operator can be used to search a dictionary by its key. There is not a convenient way to search a dictionary by value (except by looping through the elements one at a time).

Nested Structures

```python
nested = { "a": 67, "b": [12, 15, 20], "c": { "hi":
    "greeting", "goodbye": "farewell" } }
print(nested["a"])
print(nested["b"])
sum = 0
for num in nested["b"]:
    sum += num
print(sum)
print(nested["c"]["hi"])
print(nested["c"]["hi"][4])
```

Expected output:

```
67
[12, 15, 20]
47
greeting
t
```

It is common to nest dictionaries as values of dictionaries or as list elements. Also, lists can be nested in dictionaries or other lists. The primary concept to remember when working with nested structures is to remember that each variable has a type (list, dictionary, string, float, etc.) and you work with that variable in ways that are appropriate to the type. In this example, `nested` is a dictionary. To access a value in the dictionary we index the dictionary with a key: `nested["b"]` which is a list of numbers. We can iterator through a list using a `for` loop (which we do and calculate the sum of the values).
The variable `nested["c"]` is itself a dictionary so we can index it with a key: `nested["c"]["hi"]`. This variable happens to be a string so we treat it like a string. If we want to access an individual character of a string we can index it with an integer: `nested["c"]["hi"][4]` (which refers to the character “t”).

Copying Lists/Dictionaries

```
import copy
nested = { "a": 67, "b": [12, 15, 20], "c": { "hi":
    "greeting", "goodbye": "farewell" } }
other = nested # other and nested are the same dictionary
other2 = nested.copy() # other2 is copy of first level of
    value
other3 = copy.deepcopy(nested) # other3 is completely
    separate copy
```

Using the assignment operator (`=`) to copy a list or dictionary actually doesn't make a copy at all. Instead it just gives another name by which the original structure can be access. So, changing `nested` or changing `other` in this example would actually be working on the same value. The built-in `.copy()` function will copy the first level of values which is all that is needed if there are no nested structures. To make a full, multi-level copy of all embedded structures it is necessary to import the `copy` library and use its `.deepcopy` function as demonstrated.

Read Data File from URL

```
from urllib.request import urlopen # once at top

with urlopen("https://example.com/data.txt") as data:
  lines = data.read().decode('utf-8').rstrip().split("\n")
print(lines)
```

Expected output: a list whose elements are the lines of the file designated by the URL.

To read a data file located at a website you can use the `urlopen` command in the `urllib` library. The actual reading works almost exactly like reading from a local file with the exception that you must specify a decoding. In this example we read the entire file into a list as a single command.

Jupyter Markdown

```
# Big Idea
Here are things to know:
 1. It's big
 1. It's fun
 1. It's __[amazing](https://josephus.hsutx.edu)__
Other thoughts:
 - It's *cool*
 - It's **sleek**
Some code:
`
for k in lst:
    print(k)
`
```

Expected output: Big Idea
Here are things to know:

1. It's big
2. It's fun
3. It's amazing

- It's *cool*
- It's **sleek**

Some code:

```
for k in lst:
     print(k)
```

Test cells in a Jupyter Notebook can contain a variety of formatted text by using a notation called "markdown". Here is a summary of commonly used markdown commands:

# text	large heading
## text	medium heading
1. text	a numbered list element (leading space required)
– text	a bulleted list element (leading space required)
¿ text	indented text
‘text‘	preformatted text (backticks)
text*	italicized text
*text**	bold text
`__[text](URL)__`	links text to specified URL

Pandas Commands

```
import pandas as pd

# load csv as dataframe
df = pd.read_csv("https://example.com/fun.csv")
df.head(10) # show first 10 rows of dataframe
df.tail()   # show last 5 rows of dataframe

from vega_datasets import data # use vega_datasets
movies = data.movies()   # import movies data set
movies.shape             # see # of rows and columns
movies.describe()        # see summary stats
movies.corr()            # see pairwise correlations

# select one or more columns from dataframe
movies['Major_Genre']
movies[['MPAA_Rating','Major_Genre']]

# Select rows 0 to 9 skipping by 2's and columns 0, 1, 7, and 8
movies.iloc[0:10:2,[0,1,7,8]]

# select rows with PG rating
movies[movies["MPAA_Rating"]=="PG"]]

# treat all titles as strings
movies['Title'].asType(str)

# Set/reset index
movies.set_index(['Title'], inplace=True)
movies.reset_index(inplace=True)

# Drop rows or columns
movies.drop(movies.index[0:6]) # delete 1st six rows
movies.drop(['Director'], axis=1) # delete column

# group by MPAA_Rating
group_by_rating = movies.groupby("MPAA_Rating")

# show average US_Gross when grouped by rating
group_by_rating["US_Gross"].agg("mean")

# enable viewing plots in Jupyter
%matplotlib inline

# Create line chart and bar chart
group_by_rating.plot("index='Year', columns='MPAA_Rating', values='US_Gross'")
group_by_rating["US_Gross"].agg("mean").plot(kind="bar")
```

Output too complex to show.

The comments in the code provide a useful description of the Pandas commands we use in this course.

API Commands

```
import requests
import json
import pandas as pd

result = requests.get("https://api.example.com/v1/all")
print(result.status_code)
print(result.text)
all_json = result.json()
print(json.dumps(all_json, indent=4))
df = pd.read_json(result.text)
df.describe()
```

Expected output: an indented JSON string returned by the API endpoint. Followed by the result of the pandas describe() function for the same data.

The requests library makes it easy to utilize a REST API. The `get()` function returns an object that provides an HTTP status code and the retrieved text. Often the text is a JSON string. To convert the string into a Python object we just use the request library's built-in JSON encoder function (`json()`). Python's json library includes a function called `dumps()` that accepts a JSON object and converts it back to a JSON string. This is useful for "pretty printing" or for passing to functions that require a string. Wonderfully, we can read JSON strings into a pandas dataframe by using the `read_json()` function.