1 Strings

Strings are a very important data type in all languages. In Python, strings may be quoted several ways:

```
outputfule = 'output.txt'
triplequotes = """woah!
  split lines"""
print(triplequotes)
```

This also works:

```
triple_single_quotes = '''I am a string too.
I can span multiple lines!'''
print(triple_single_quotes)
```

Quotes in quotes In Python, we can quote strings with either single ('') or double quotes (""). Sometimes we want to create a string that contains quotes. This is easy to do! Strings quoted with '' can contain "":

```
'Bob said, "it is hot out there today".'
```

Strings quoted with "" can contain '':

```
"Python, it's a wonderful language"
```

Or we can escape the quote with ':

```
'it\'s not Nick\'s birthday today'
```

```
"I don't always quote my strings, but when I do, I prefer \""
```

1.1 Strings vs. Numbers

```
a = 5
b = '5'
a + b
```

```
print("type(a): ", type(a))
print("type(b): ", type(b))
```

It is simple to convert between numbers and strings!

```python
# convert int to a string
a = str(55)
print(a)
print(type(a))
```

```python
# convert string to a float
a = float("99.45")
print(a)
print(type(a))
```

1.2 Slicing

```python
quote = """That's all folks!"""
print(quote[2])
print(quote[7:10])
print(quote[:4])
print(quote[7:])
print(quote[:-7])
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered ‘0‘. Then the right edge of the last character of a string of ‘n‘ characters has index ‘n‘, for example:

```python
word = 'Python'
```

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

```python
word[-2:]
```

1.3 Strings are immutable

We can access an individual character of a string:

```python
a = 'hello'
a[0]
```

We cannot change any part of a string:

```
a[0] = 'k'
```

1.4 Concatenation

Concatenate (add together) strings with '+':

```
b = 'j' + a[1:]
b
```

This creates a new string.

1.5 String functions / methods

```
name = 'Leland'
len(name)
name.lower()
name.upper()
name.find('lan')
name.find('lan', 1, 4)
```

There are many string methods

1.6 String formatting

It is often important to create strings formatted from a combination of strings, numbers, and other data. In Python 3 this is best handled by the 'format' string method. Here is a simple example:

```
name = "Nick"
course = 'CME211'
print("My name is {0}. I am the instructor for {1}.".format(name,course))
```

Format strings contain "replacement fields" surrounded by curly braces ''. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: '' and ''. The number in the braces refers to the order of arguments passed to 'format'. Numbers don't need to be specified if the sequence of braces has the same order as arguments:

```
program = 'CME'
number = 211
print("this course is: {}-{}".format(program,number))
```

String formatting is a good way to combine text and numeric data.

String formatting is also how we control the output of floating point numbers:

```python
print("     {{:f}}: {:f}".format(42.42))
print("     {{:g}}: {:g}".format(42.42))
print("     {{:e}}: {:e}".format(42.42))
print("   {{:.2e}}: {:.2e}".format(42.42))
print("{{: 8.2e}}: {: 8.2e}".format(42.42))
print("{{: 8.2e}}: {: 8.2e}".format(-1.0))
```

See the Python Format Mini-Language docs and more examples

1.7 Numeric operations in Python

The following operations are defined for numeric types (i.e. `int` and `float`):

- `x + y`: sum of `x` and `y`
- `x - y`: difference of `x` and `y`
- `x * y`: product of `x` and `y`
- `x / y`: quotient of `x` and `y`
- `x // y`: floored quotient of `x` and `y`
- `x % y`: remainder of `x / y`
- `-x`: `x` negated
- `+x`: `x` unchanged
- `abs(x)`: absolute value or magnitude of `x`
- `int(x)`: `x` converted to integer
- `float(x)`: `x` converted to floating point
- `x ** y`: `x` to the power `y`

A complete list of operations and more detailed discussion can be found in the Python documentation.

1.7.1 Complex numbers

Python has built-in complex numbers:

```python
c = 3 + 6j
print(c)
print(type(c))
```

Access real and imaginary parts:

```python
print(c.real)
print(c.imag)
```

The parts have type `float`:

```python
print(type(c.real))
print(type(c.imag))
```

1.7.2 Numeric conversions

It is often useful to convert between integers and floating point numbers. Python fully supports mixed arithmetic: when a binary arithmetic operator (such as `+` or `*`) has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point. Comparisons between numbers of mixed type use the same rule. The constructors `int()` and `float()` can be used to produce numbers of a specific type.

Let's see some examples:

```python
x = 1 + 2.0
print(x)
print(type(x))
```

In this case the integer `1` is "widened" or converted to the floating point number `1.0` before the addition.

It is possible to manually convert from `int` to `float`:

```python
x = float(3)
print(x)
print(type(x))
```

It is also possible to convert from `float` to `int`:

```python
x = int(4.7) print(x) print(type(x))
```

Let's see what happens with negative numbers:

```
x = int(-8.9) print(x) print(type(x))
```

The Python `int()` constructor rounds floating point numbers towards `0`.

1.7.3 Converting to and from strings

Python makes it very easy to convert numbers to and from strings. This is a useful feature when trying to read numbers from a text file. Let's see it in action:

```
my_num_str = "42" print(type(my_num_str))  my_num_int = int(my_num_str) my_num_floa
float(my_num_str)  print(my_num_int) print(type(my_num_int))  print(my_num_float) p
```

It also easy to convert from a numeric type back to a string:

```
print("exam score:" + str(95) + "%")
```

An attempt to concatenate a string with a numeric type is an error:

```
print("exam score:" + 95 + "%")
```

It is better to use string formatting for this:

```
print("exam score: {}%".format(95))
```

1.8 Lists

Sequential containers store data items in a specified order. Think elements of a vector, names in a list of people that you want to invite to your birthday party. The most fundamental Python data type for this is called a `list`. Later in the course we will learn about containers that are more appropriate (and faster) for numerical data that come from NumPy.

1.8.1 Creating lists

In Python, lists are created by putting things inside of square brackets (`[]`).

```
some_primes = [1,2,3,5,7,11] print(some_primes)
```

Lists can hold objects of any type:

```
many_types = [1, 55.5, "Am I in a list?", True] print(many_types)
```

We can get the length of the list with the `len()` functions:

```
len(many_types)
```

1.8.2 Accessing Elements

Elements of a list are accessed using square brackets after a variable.

```
myList = [5, 2.3, 'hello']
```

The first element of a list is at index `0`:

```
mylist[0]
```

```
myList[2]
```

Attempting to access an element out of bounds will produce an error:

```
mylist[3]
```

We can index to `-1` to get the object at the end of the list:

```
myList[-1]
```

Likewise, we can index backwards using negative numbers:

```
myList[-3]
```

1.8.3 Slicing

A sub-list may be accessed using slice syntax. Let's start with the list:

```
many_types = [1, 55.5, "Am I in a list?", True, "the end"]
```

Let's look at a sub-list:

```
many_types[2:4]
```

The `[2:4]` is called a slice and returns a list with elements at indices 2 and 3 from the original list.

It is easy to slice from an index to the end:

```
many_types[2:]
```

It is also easy to slice from the beginning to a specified index:

```
many_types[:3]
```

1.8.4 Adding and multiplying

The + operator concatenates (or combines) lists:

```
myList = [5, 2.3, 'hello'] mySecondList = ['a', '3'] concatList = myList +
mySecondList print(concatList)
```

The * operator can be used to repeat lists:

```
myList = ['hello', 'world'] print(myList * 2) print(2 * myList)
```

1.8.5 Lists are mutable

Lists are mutable, this means that individual elements can be changed.

```
# start with a list my_list = ['a', 43, 1.234] # assign a new value to index 0 my_l
```

We can also assign to a slice

```
x = 2 my_list[1:3] = [x, 2.3] print(my_list)
```

1.8.6 Copying a list

Let's attempt to copy a list referenced by variable `a` to another variable `b`:

```
a = ['a', 'b', 'c'] b = a # attempt to copy a to b b[1] = 1 # now we want to change
```

That's interesting! Modifying `b` caused `a` to change.

Let's look at this example in Python Tutor.

Ok, let's try a different technique.

```
 = ['a', 'b', 'c'] b = a        # first attempt to copy a to b c = list(a) # use the
# now we want to change an element in b print("a: ", a) print("b: ", b) print("c: "
```

Again, let's try this example in Python Tutor.

A list can be copied with `b = list(a)` or `b = a[:]`. The second option is a slice including all elements.

1.8.7 Python's data model

Variables in Python are actually a reference to an object in memory. Assignment with the `=` operator sets the variable to refer to an object. Here is a simple example to demonstrate this property:

```python
a = [1,2,3,4] b = a b[1] = 55 print(b) print(a)
```

In this example, we assigned `a` to `b` via `b = a`. This did not copy the data, it only copied the reference to the list object in memory. Thus modifying the list through `b` also changes the data that you will see when accessing from `a`. You can inspect object ids in Python with:

```python
print("id(a): ", id(a)) print("id(b): ", id(b))
```

Those numbers refer to memory addresses.

1.8.8 Copying data (more generally)

The `copy` function in the `copy` module is a more generic way to copy a list:

```python
import copy a = [5,2,7,0,'abc'] b = copy.copy(a) b[4] = 'xyz' print(b) print(a)
```

Note that elements in a list are also references to memory location. For example if your list contains a list, this will happen when using `copy.copy()`:

```python
a = [2, 'string', [1,2,3]] b = copy.copy(a) b[2][0] = 55 print(b) print(a)
```

Here, the element for the sub-list `[55, 2, 3]` is actually a memory reference. So, when we copy the outer list, only references for the contained objects are copied. Thus in this case modifying the copy (`b`) modifies the original (`a`). Thus, we may need the function `copy.deepcopy()`:

```python
a = [2, 'string', [1,2,3]]
b = copy.deepcopy(a)
b[2][0] = 99
print(b)
print(a)
```

1.8.9 Sorting lists

Sorting Python lists is very easy. Let's randomly shuffle a list and then sort it.

```
import random my_list = list(range(10)) print(my_list) random.shuffle(my_list) prin
```

Note that the `random.shuffle()` function shuffles the list in-place. It does not create a new list.

We can use the `sorted()` function to return a new sorted list:

```
sorted_list = sorted(my_list) print("my_list:", my_list) print("sorted_list:", sort
```

The list `sort()` method sorts the list in place:

```
my_list.sort() print("my_list:", my_list)
```

1.8.10 List operations

In the following summary, `s` is a list and `x` is an element.

- `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` or `n * s`: equivalent to adding `s` to itself `n` times
- `s[i]`: ith 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`
- `s.index(x)`: index of the first occurrence of `x` in `s`
- `s.count(x)`: total number of occurrences of `x` in `s`
- `s[i] = x`: item `i` of `s` is replaced by `x`
- `s[i:j] = t`: slice of `s` from `i` to `j` is replaced by the contents of the `t`

- `del s[i:j]`: same as `s[i:j] = []`
- `s[i:j:k] = t`: the elements of `s[i:j:k]` are replaced by those of `t`
- `del s[i:j:k]`: removes the elements of `s[i:j:k]` from the list
- `s.append(x)`: appends `x` to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
- `s.clear()`: removes all items from `s` (same as `del s[:]`)
- `s.copy()`: creates a shallow copy of `s` (same as `s[:]`)
- `s.extend(t) or s += t`: extends `s` with the contents of `t` (for the most part the same as `s[len(s):len(s)] = t`)
- `s *= n`: updates `s` with its contents repeated `n` times
- `s.insert(i, x)`: inserts `x` into `s` at the index given by `i` (same as `s[i:i] = [x]`)
- `s.pop([i])`: retrieves the item at `i` and also removes it from `s`
- `s.remove(x)`: remove the first item from `s` where `s[i] == x`
- `s.reverse()`: reverses the items of `s` in place
- `s.sort()`: sorts the items of `s` in place

Also have a look at `help(list)`.

1.8.11 Resources

- Python tutorial section on `list`
- Python reference section on sequences

1.9 Looping with `for` and `while`

Very often, one wants to repeat some action. This can be achieved with `for` and `while` statements.

1.9.1 `for` loops

A `for` loop is typically used when we want to repeat an action a given number of times.

```
for i in range(5):     print(i**2, end=', ') print() # print a new line at the end
```

Here, `for i in range(n):` will execute the loop body `n` times with `i = 0, 1, 2, ..., n - 1` in succession.

1.9.2 Note on Python syntax

Python uses syntatic indenting. This means that indenting code has a meaning in the programming language. In languages like C, C++, and Java, loop bodies are enclosed in braces, but good coding style suggests that statements in a loop or conditional body are indented:

```
for (int i = 0; i < 10; i++) {     printf("i = %d\n",i); }
```

Python takes this a step further and requires the indenting of loop and conditional bodies. We recommend that you use 4 spaces to indent python code (so does the python community]). Please tell your text editors to insert spaces instead of tab characters when you hit the tab key on the keyboard.

1.9.3 The `range()` function

The `range()` function can be used in a few different ways. We can convert a range object to a python list with the `list()` function:

```
# get range 0,...,6 print(list(range(7))) # get range 4,...,10 print(list(range(4,1
```

See `help(range)` for more info.

Note that `range` differs in Python 2 and 3. In Python 2, `range()` returns a list. In Python 3, `range()` returns an object that produces a sequence of integers in the context of a `for` loop, which is more efficient, because memory for a new list need not be allocated.

1.9.4 `for` and lists

We can use a `for` loop to iterate over items in a list:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]] for item in my_list:
print(item)
```

It is often handy to get access to both the list item and index in a `for` loop. This can be achieved with the `enumerate()` function:

```
my_list = [1, 45.99, True, "str item", ["sub", "list"]] for index, item in enumerat
print("{}: {}".format(index, item))
```

1.9.5 Example adding numbers

```
summation = 0 for n in range(1,101):     summation += n print(summation)
```

Also achievable in Python via `sum`:

```
sum(range(1,101))
```

1.9.6 `while` loops

When we do not know how many iterations are needed, we can use `while`.

```
i = 2 while i < 100:     # loop body only execute if conditional statement is True
print(i**2,end=", ")     i = i**2 print() # print a new line at the end
```

1.9.7 Infinite loops

```
while True:     print("hah!")
```

- In Jupyter Notebook, select “Interrupt” from the Kernel menu
- Use `ctrl-c` to interrupt the interpreter

1.9.8 Nesting loops

A *nested loop* is a loop in the body of a loop.

```
for i in range(8):     for j in range(i):          print(j, end=' ')
print()
```

1.9.9 continue

`continue` continues with the next iteration of the smallest enclosing loop:

```
\begin{Highlighting}[]
\ControlFlowTok{for} \NormalTok{num }\OperatorTok{in} \BuiltInTok{range}\NormalTok{
    \ControlFlowTok{if} \NormalTok{num }\OperatorTok{%} \DecValTok{2} \OperatorTok{
        \BuiltInTok{print}\NormalTok{(}\StringTok{"Found an even number:"}\NormalTo
        \ControlFlowTok{continue}
    \BuiltInTok{print}\NormalTok{(}\StringTok{"Found an odd number:"}\NormalTok{, n
for num in range(2, 10):     if num % 2 == 0:         print("Found an even number:"
continue     print("Found an odd number:", num)
```

Here, `num in range(2,10)` sets up a loop where `num = 2, 3, ..., 9`.

1.9.10 break

The `break` statement allows us to jump out of the smallest enclosing `for` or `while` loop.

Finding prime numbers:

```
max_n = 10 for n in range(2, max_n):     for x in range(2, n):
if n % x == 0: # n divisible by x            print(n, 'equals', x, '*', n/x)
break     else: # executed if no break in for loop          # loop fell through with
print(n, 'is a prime number')
```

1.9.11 pass

The `pass` statement does nothing, which can come in handy when you are working on something and want to implement some part of your code later.

```
traffic_light = 'green' if traffic_light == 'green':     pass # to implement later
print('whatever you do, stop the car!')
```

1.9.12 Loop else

- An `else` can be used with a `for` or `while` loop
- The `else` is only executed if the loop runs to completion, not when a `break` statement is executed

```
for i in range(4):     print(i) else:     print("all done")   for i in range(7):
print(i)     if i > 3:         break else:     print("all done")
```

1.9.13 A note on Python variables

It is bad practice to define a variable inside of a conditional or loop body and then reference it outside:

```
name = "Nick" if name == "Nick":      age = 45 # newly created variable
print("Nick's age is {}".format(age))
```

Here is what happens when a variable is not created:

```
name = "Bob" if name == "Nick":      id_number = 45 # also newly created variable
print("{}'s id number is {}".format(name, id_number))
```

Good practice to define/initialize variables at the same level they will be used:

```
name = "Bob" age = 55 if name == "Nick":      age = 45  print("{}'s age is {}".forma
```

1.10 Python File IO (Input-Output)

Python makes it very easy to read and write files to disk.

Keep in mind that it is almost always better to use a Python module for specific formats. For example, use the `json` module for JSON files or the `csv` module for `.csv` files. Better yet, use Pandas for table-like data.

1.10.1 What is a file?

A *file* is a segment of data, typically associated with a filename, that exists in a computer's persistent storage. This means that the data remains when the computer is turned off.

There are two main kinds of files: *text* and *binary*.

Text files are typically easier for humans to read and write.

Binary files (images, music files, etc.) are more efficient in terms of storage.

Python scripts are text files and by convention have a `.py` extension. On unix systems we can dump a text file to the terminal with:

```
$ cat hello.py
# run me from the command line with
# $ python3 hello.py

print("hello sweet world!")
```

For fun, try dumping a binary file to the terminal with `$ cat /bin/ls`. What happens?

In Python it is very easy to open, read from, and write to a text file. Let's see how it works.

See Chapter 9 in **Learning Python** for information on accessing files with Python. The relevant information starts on page 282.

1.10.2 The file object

- Interaction with the file system is pretty straightforward in Python.
- Done using *file objects*
- We can instantiate a file object using `open` or `file`

1.10.3 Opening a file

```
f = open(filename, option)
```

- `filename`: path to file on disk
- `option`: mode to open file (passed as a string)
- `'r'`: read file
- `'w'`: write to file (overwrites existing file)
- `'a'`: append to file
- We need to close a file after we are done: `f.close()`

Open a file:

```
f = open("humpty-dumpty.txt","r") f
```

We can test if the file is closed:

```
f.closed
```

We can close the file:

```
f.close() f.closed
```

Closing a file flushes any buffered data to disk and frees up operating system resources. If using a file in this manner, it is important to close files. *We will take off points if you neglect to do this.*

1.10.4 `with open() as f:`

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.

```
with open('humpty-dumpty.txt', 'r') as f:      print(f.read())
f.closed
```

1.10.5 If a file does not exist

```
bad_file = open("no-file.txt","r")
```

1.10.6 Reading data from a file

File object methods:

- `read()`: Read entire file (or first `n` characters, if supplied)
- `readline()`: Reads a single line per call
- `readlines()`: Returns a list with lines (splits at newline)

1.10.7 `readline()`

Use the `readline()` method to read lines from a file:

```
f = open("humpty-dumpty.txt","r") print(f.readline()) print(f.readline()) f.close()
```

1.10.8 `read()`

You can read an entire file at once with the `read()` method:

```
f = open("humpty-dumpty.txt","r") poem = f.read() print(poem) f.close()
```

1.10.9 Iterate over lines

You can very easily iterate over lines in a file with:

```
f = open("humpty-dumpty.txt","r") for line in f:     print(line) f.close()
```

1.10.10 An example with `with`

```
with open('humpty-dumpty.txt', 'r') as f:     for i, line in enumerate(f):
print("line {}: {}".format(i,line))
```

Note the extra lines between each line of text. You can do this by specifying the `end` keyword parameter for the `print` function to be an empty string (`""`): `print(line, end='')` or slicing `line` with `print(line[:-1])`.

1.10.11 Iterate over words!

The string `split` method partitions a string into a list based on a delimiter. Space is the default delimiter. The `strip` method removes leading and trailing whitespace from a string.

```
f = open("humpty-dumpty.txt","r") for line in f:     for word in line.split():
# use strip() method to remove extra newline characters         print(word.strip())
```

1.10.12 Writing to file

Use the `write()` method to write to a file. Make sure to open the file in write mode with `'w'` as the second argument to `open()`.

```
name = "Python learner" with open('hello.txt', 'w') as f:     f.write("Hello, {}!\n
```

`cat hello.txt}`

1.10.13 More writing examples

Write elements of list to file:

```
xs = ["i", "am", 'a', 'fancy', 'list', 42] with open('from_list.txt', 'w') as f:
for x in xs:         f.write('{}\n'.format(x))
```

```
\OperatorTok{%}\NormalTok{cat from_list.txt}
```

To write multiple lines to a file at once, use the `writelines` method:

```
f = open("writelines.txt","w") f.writelines(["a mighty fine day\n","ends with a gre
```

```
\OperatorTok{%}\NormalTok{cat writelines.txt}
```

Note that the `write` and `writelines` methods will not insert newline characters. To get a new line, you must add `'\n'` to the strings.

1.10.14 Buffering

For efficiency, the `file` object will temporarily store data from `write` or `writelines` methods in memory before actually writing to disk. This is known as buffering. It turns out that writing larger chunks of data to disk in fewer transactions is more efficient than many transactions of small chunks. If you attempt to open a text file created by Python and not closed, you may not see the data. Calling the `close()` method flushes all data to disk.

```
f = open('foo.txt','w') f.write("this is some text\n")
```

(On my system `foo.txt` is empty at this point. Behavior may be different on your system.)

```
f.close()
```