Introduction to Python

COMP7023 Predictive Analytics

Rosalind Wang

Why Python?

- Conceived in the 1980s as a teaching and scripting language, Python has become extremely popular in the past decade.
- One of two "must know" languages for data scientists.
- A large ecosystem of domain-specific tools
- For scientific programming and data science, we have:
    - NumPy: storage and computation for multidimensional data arrays
    - SciPy: numerical tools, e.g. integration, interpolation
    - Pandas: for dataframe objects and related methods for data manipulation
    - Matplotlib: plots and figures
    - Scikit-Learn: common machine learning algorithms
    - IPython/Jupyter: enhanced terminal and interactive notebook environment.

Books and resources

There are many books and resources online for learning Python.

This set of notes will follow closely to the book *A Whirlwind Tour of Python* by Jake VanderPlas, O'Reilly, 2016

Installation

I recommend install Python through Anaconda (https://www.anaconda.com), which comes with a bundle of packages for scientific programming.

The Zen of Python

To see the programming philosophy behind Python, run `import this`:

In [1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

How to run Python code

Through Python interpreter, i.e. via command prompt on the Terminal:

```
python
Python 3.6.8 |Anaconda, Inc.| (default, Dec 29 2018, 19:04:46)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

An IDE like `Spyder`

Jupyter notebook: an interactive terminal for code, text, output.

A Quick Tour of Python Language Syntax

Python has very clean syntax.

Leading to some calling it "executable pseudocode".

We'll discuss the main features of Python's syntax first.

Consider the following code example:

In [2]:
```
# set the midpoint
midpoint = 5

# make two empty lists
lower = []; upper = []

# split the numbers into lower and upper
for i in range(10):
    if (i < midpoint):
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)
```

```
lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

Basic Semantics: Variables and Objects

Variables and objects are the main ways you store, reference and operate on data in Python.

Variables are pointers

Assign variables in Python using `=`

```
# assign 4 to the variable x
x = 4
```

Defining a *pointer* named `x` that points to some other bucket containig the value `4`.

Consequence of this: no need to declare the variable, or require the variable to point to the same type.

Python is *dynamically typed* : variable names can point to object of any type. e.g.

```
x = 1          # x is an integer
x = 'hello'    # now x is a string
x = [1, 2, 3] # now x is a list
```

Warning: variables are pointers

If we have two variables pointing to the same *mutable* object, then changing one will change the other:

In [3]:
```
x = [1, 2, 3]
y = x
```

In [4]:
```
print(y)
```

```
[1, 2, 3]
```

In [5]:
```
x.append(4)   # append 4 to the list points to be x
print(y)      # print out what's inside y
```

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

Note if we use `=` to assign another value to `x`, this will not affect the value of `y`

Assignment changes what object `x` now points to.

In [6]:
```
x = 'something else'
print(y)
```

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

Everything is an Object

Python is an object-oriented programming language and everything in Python is an object.

An *object* is an entity that contains data along with associated metadata and/or functionality.

"Everything is an object" means every entity has some metadata (called attributes) and associated functionality (called methods).

These attributes and methods are accessed via the dot syntax.

In [7]:
```
L = [1, 2, 3]
L.append(100)
print(L)
```

```
[1, 2, 3, 100]
```

Even simple types have attached attributes:

In [8]:
```
x = 4.5
print(x.real, "+", x.imag, 'i')
```

```
4.5 + 0.0 i
```

Even the attributes and methods are objects with their own `type` information:

In [9]:
```
type(x.is_integer)
```

Out[9]:
```
builtin_function_or_method
```

Basic Semantics: Operators

Arithmetic operations:

There are eight basic binary operators:

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a – b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`–a`	Negation	The negative of `a`
`+a`	Unary plus	`a` unchanged (rarely used)
`a @ b`	Matrix product	Matrix product of `a` and `b`

In [10]:
```
# addition, subtraction, multiplication
(4 + 8) * (6.5 - 3)
```

Out[10]:
```
42.0
```

In [11]:
```
# True division
print(11 / 2)
```

```
5.5
```

In [12]:
```
# Floor division
print(11 // 2)
```

```
5
```

Bitwise operations:

For bitwise logical operations on integers.

Most widely used are:

- `a & b` for bitwise AND
- `a | b` for bitwise OR

Exercise to find out about the rest.

Assignment operations:

We can combine assignment operator `=` with any of the operators mentioned earlier:

```
a += b          a -= b          a *= b          a /= b
a //= b         a %= b          a **= b         a &= b
a |= b          a ^= b          a <<= b         a >>= b
```

For any operator `#`, the expression `a #= b` is the same as `a = a # b`.

Note: for mutable objects like lists, arrays or data frames, these operators modify the contents of the original object ranter than creating a new object to store the result.

Comparison Operations

When we want to compare different values.

Python implements standard comparison operators, which returns Boolean values `True` and `False`.

Operation	Description
`a == b`	a equal to b
`a != b`	a not equal to b
`a < b`	a less than b
`a > b`	a greater than b
`a <= b`	a less than or equal to b
`a >= b`	a greater than or equal to b

Comparison operators can be combined with arithmetic and bitwise operators:

In [13]:
```
# 25 is odd
25 % 2 == 1
```

Out[13]:
```
True
```

In [14]:
```
# 66 is odd
66 % 2 == 1
```

Out[14]:
```
False
```

In [15]:
```
# multiple comparisons:
# check if a is between 15 and 30
a = 25
15 < a < 30
```

Out[15]:
```
True
```

Boolean operations

When working with Boolean values we can use the standard concepts of "and", "or" and "not".

The operators are `and`, `or`, and `not`:

In [16]:
```
x = 4
(x < 6) and (x > 2)
```

Out[16]:
```
True
```

In [17]:
```
(x > 10) or (x % 2 == 0)
```

Out[17]:
```
True
```

In [18]:
```
not (x < 6)
```

Out[18]:
```
False
```

Boolean operations are extremely useful in *control flow statements*, which we'll discuss soon.

Identity and membership operators

Prose-like operators to check for identiy and membership

Operator	Description
`a is b`	True if `a` and `b` are identical objects
`a is not b`	True if `a` and `b` are not identical objects
`a in b`	True if `a` is a member of `b`
`a not in b`	True if `a` is not a member of `b`

In [19]:
```
1 in [1, 2, 3]
```

Out[19]:
```
True
```

Build-in Types: Simple values

Type	Example	Description
`int`	`x = 1`	Integers (i.e., whole numbers)
`float`	`x = 1.0`	Floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with a real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

Build-in Data Structure

Compound type, acting as containers for other types:

Type Name	Example	Description
`list`	`[1, 2, 3]`	Ordered collection
`tuple`	`(1, 2, 3)`	Immutable ordered collection
`dict`	`{'a':1, 'b':2, 'c':3}`	Unordered (key,value) mapping
`set`	`{1, 2, 3}`	Unordered collection of unique values

Note the different brackets have distinct meanings

Lists

Basic *ordered* and *mutable* data collation type in Python.

Defined with comma-separated values between square brackets

In [1]:
```
# First few prime numbers in a list:
L = [2, 3, 5, 7]
```

In [134]:
```
# Can also contain a mix of types
L2 = [1, 'two', 3.14, [0, 3, 5]]
```

Lists have a number of useful properties and methods:

In [2]:
```
# Length of a list
len(L)
```

Out[2]:
```
4
```

In [3]:
```
# Append a value to the end
L.append(11)
L
```

Out[3]:
```
[2, 3, 5, 7, 11]
```

In [4]:
```
# Addition concatenates lists
L + [13, 17, 19]
```

Out[4]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

In [138]:
```
# sort() method sorts in-place
L = [2, 5, 1, 6, 3, 4]
L.sort()
L
```

Out[138]:
```
[1, 2, 3, 4, 5, 6]
```

List indexing and slicing

Access to elements in compound types is through

- *indexing* for single elements
- *slicing* for multiple elements

Indexing

In [5]:
```
L = [2, 3, 5, 7, 11]

L[0]    # zero-based indexing
```

Out[5]: 2

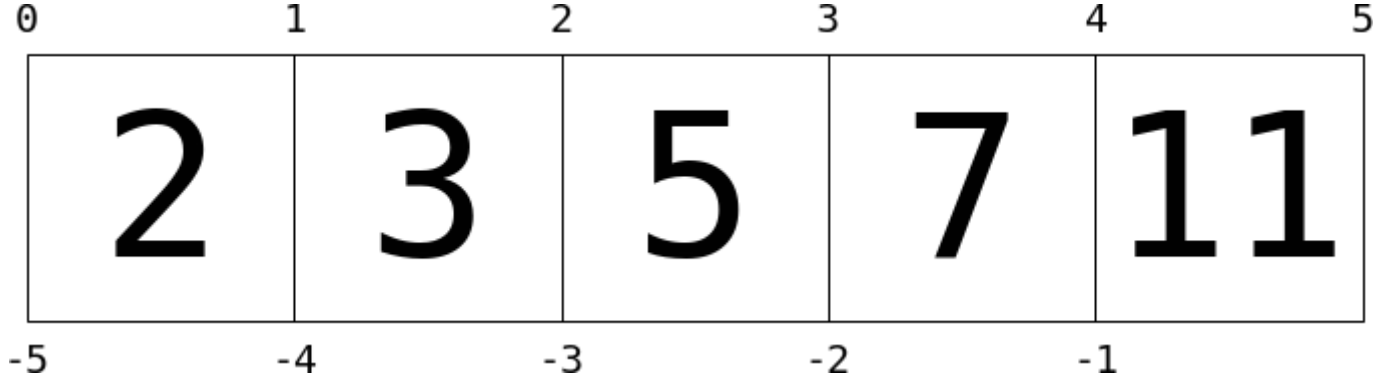
In [6]:
```
L[1]
```

Out[6]: 3

In [7]:
```
# get the element at the end of the list
L[-1]
```

Out[7]: 11

Think of it this way:



Values are represented by large numbers in the squares

List indices are represented by small numbers above and below

- `L[2]` returns 5
- `L[-2]` returns 7

A note about indexing

Since Python's index start from `0`, for all materials in this subject we'll use the following language:

- element n of variable x: `x[n]`
- n-th element of variable x: `x[n-1]`

For example, given the list in the previous slide, then:

In [8]:
```
# get element 2 from list L:
L[2]
```

Out[8]: 5

In [9]:
```
# get 2nd element from list L:
L[1]
```

Out[9]: 3

Slicing

Slicing allows accessing of multiple values in sublists:

In [29]:
```
# To get the first three elements of the list
L[0:3]
```

Out[29]:
```
[2, 3, 5]
```

In [30]:
```
# We can leave out the first index, if we want to start from the beginning
L[:3]
```

Out[30]:
```
[2, 3, 5]
```

In [31]:
```
# Similarly, if we leave out the last index
L[-3:]
```

Out[31]:
```
[5, 7, 11]
```

It is possible to specify a third integer that represents the step size:

In [32]:
```
L[::2]          # equivalent to L[0:len(L):2]
```

Out[32]:
```
[2, 5, 11]
```

In [33]:
```
# Reverse the array
L[::-1]
```

Out[33]:
```
[11, 7, 5, 3, 2]
```

Tuples

Similar to lists, but defined with parentheses:

In [34]:
```
t = (1, 2, 3)
```

In [35]:
```
# or without the brackets
t = 1, 2, 3
print(t)
```

```
(1, 2, 3)
```

Tuples have a length, and individual element can be extracted:

In [36]:
```
len(t)
```

Out[36]: 3

In [37]:
```
t[0]     # notice the square bracket here
```

Out[37]: 1

Tuples are *immutable* : once created, they can't be changed

In [38]:
```
t[1] = 4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-87b0f225887f> in <module>
----> 1 t[1] = 4

TypeError: 'tuple' object does not support item assignment
```

In [41]:
```
t.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-41-ada7ed8a579e> in <module>
----> 1 t.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

Tuples are often used: a common case is multiple return values in functions:

In [63]:
```
x = 0.125
x.as_integer_ratio()
```

Out[63]:
```
(1, 8)
```

In [64]:
```
# multiple return values can be individually assigned:
numerator, denominator = x.as_integer_ratio()
print(numerator / denominator)
```

```
0.125
```

Dictionaries

Used to map keys to values.

Created via a comma-separated list of `key:value` pairs:

In [65]:
```
numbers = {'one':1, 'two':2, 'three':3}
```

In [66]:
```
# Accessing a value via the key
numbers['two']
```

Out[66]: 2

In [67]:
```
# Add new items
numbers['ninty'] = 90
print(numbers)
```

```
{'one': 1, 'two': 2, 'three': 3, 'ninty': 90}
```

Sets

Unordered collections of unique items

In [68]:
```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
```

Set operations follows the mathematics of sets, e.g. unions, intersection, difference, etc.

In [69]:
```
# union: items appearing in either
primes | odds          # with an operator
primes.union(odds)     # equivalent with a method
```

Out[69]:
```
{1, 2, 3, 5, 7, 9}
```

In [70]:
```
# intersection: items appearing in both
primes & odds                # with an operator
primes.intersection(odds)    # equivalent with a method
```

Out[70]:
```
{3, 5, 7}
```

Control Flow

To execute certain code blocks conditionally and/or repeatedly

We'll cover conditional and loop statements.

Conditional Statements: if, elif and else

Allow programer to execute certain pieces of code depending on some Boolean condition.

In [71]:
```
x = -15

if x == 0:
    print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")
```

```
-15 is negative
```

for loops

To repeately execute some code statement.

In [72]:
```
for N in [2, 3, 5, 7]:
    print(N, end=' ') # print all on same line
```

```
2 3 5 7
```

In [73]:
```
# using a simple iterator
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

while loops

Iterates until some condition is met:

In [74]:
```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

break and continue

To fine-tune how the loops are executed:

- `break` : breaks out of the loop entirely
- `continue` skips the remainder of the current loop, and goes to the next iteration

Can be used in both `for` and `while` loops.

Functions

Modular codes make your program more readable and reusable

Two ways of creating functions:

- `def` to create any type of function
- `lambda` to create short anonymous function

Defining functions

Using the `def` statement:

In [75]:
```
# create a function to generate the fibonacci sequence
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

In [76]:
```
fibonacci(10)
```

Out[76]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Default argument

Values that we want the function to use *most* of the time, but we also like to give the users some flexibility.

In [77]:
```
# use default arguments
def fibonacci(N, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

In [78]:
```
# Using just one argument
fibonacci(10)
```

Out[78]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [79]:
```
# But if we want to start from a different position
fibonacci(10, b=3, a=1)
```

Out[79]:
```
[3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

Felxible arguments

When you don't initially know how many arguments the user will pass.

Use the special form `*args*` and `**kwargs**` to catch all arguments:

In [80]:
```
def catch_all(*args, **kwargs):
    print("args =", args)
    print("kwargs = ", kwargs)
```

In [81]:
```
catch_all(1, 2, 3, a=4, b=5)
```

```
args = (1, 2, 3)
kwargs =  {'a': 4, 'b': 5}
```

In [82]:
```
catch_all('a', keyword=2)
```

```
args = ('a',)
kwargs =  {'keyword': 2}
```

Anonymous functions

Short, one-off functions can be defined with the `lambda` statement:

In [83]:
```
add = lambda x, y : x+y
add(1,2)
```

Out[83]:
```
3
```

This is roughly equivalent to:

```
def add(x, y):
    return x+y
```

Why use `lambda` functions?

- Everything in Python is an object, even functions
- So functions can be passed as arguments to functions -- functional programming
- Allows concise code, when we only need a short function
- Sometimes we just need a throwaway function

Errors and Exceptions

Coding mistakes are inevitable, they come in three basic flavours:

Syntax errors:

- where the code is not valide Python

Runtime errors:

- where syntactically valid code fails to execute, perhaps due to invalide user input

Semantic errors:

- errors in logic: code executes without a problem, but the result is not what you expect

The first two are often easy to fix, but the third is often difficult to identify and fix.

Runtime errors

They can happen in a lot of ways:

In [84]:
```
# reference an undefined variable:
print(Q)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-84-7255616892d6> in <module>
      1 # reference an undefined variable:
----> 2 print(Q)

NameError: name 'Q' is not defined
```

In [85]:
```
# operation that's not defined:
1 + 'abc'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-394033ccfa4c> in <module>
      1 # operation that's not defined:
----> 2 1 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [86]:
```
# accessing element that doesn't exist
L = [1, 2, 3]
L[1000]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-86-b79cd14edc9d> in <module>
      1 # accessing element that doesn't exist
      2 L = [1, 2, 3]
----> 3 L[1000]

IndexError: list index out of range
```

Notice Python gives you *meaningful* exception that includes information about what went wrong.

This is a case of practice makes perfect: learn to read and understand the error messages.

Catch and raise exceptions:

`try` ... `except` clause allows you to handle runtime exceptions.

`raise` statement allows you to raise your own exceptions.

Found out more about these in the book.

Iterators

Objects you can *iterate* through.

Previously, we've seen the `range` iterator:

In [87]:
```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Iterating over lists

Perhaps the easiest way to understand iterators is through lists:

In [88]:
```
for value in [2, 4, 6, 8, 10]:
    # do some operation
    print(value + 1, end = ' ')
```

```
3 5 7 9 11
```

The `for x in y` syntax allows us to repeat some operation for each value in the list.

Useful iterator: enumerate

When you not only iterate through the values in an array, but also keep track of the index:

In [89]:
```
# you might be tempted to do this:
L = [2, 4, 6, 8, 10]
for i in range(len(L)):
    print(i, L[i])
```

```
0 2
1 4
2 6
3 8
4 10
```

In [90]:
```
# a better option using enumerate
for i, val in enumerate(L):
    print(i, val)
```

```
0 2
1 4
2 6
3 8
4 10
```

Useful iterator: zip

Iterating multiple lists simultaneously:

In [91]:
```
L = [2, 4, 6, 8, 10]
R = [3, 6, 9, 12, 15]
for lval, rval in zip(L, R):
    print(lval, rval)
```

```
2 3
4 6
6 9
8 12
10 15
```

List comprehensions

Compact ways of constructing lists using just a single line of code:

In [92]:
```
[i for i in range(20) if i % 3 > 0]
```

Out[92]:
```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Creating a list of numbers that excludes multiples of 3. Without list comprehension, this might be 10 lines of code.

Basic list comprehension

Compress a list building `for` loop into a single short, readable line.

For example, a loop that constructs a list of first 12 square integers:

In [93]:
```
L = []
for n in range(12):
    L.append(n ** 2)
L
```

Out[93]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The list comprehension equivalent is:

In [94]:
```
[n ** 2 for n in range(12)]
```

Out[94]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The basic syntax is therefore `[expr for var in iterable]`

Mutliple iteration

We can also build a list from two values, simply add another `for` expression:

In [95]:
```
[(i, j) for i in range(2) for j in range(3)]
```

Out[95]:
```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Conditional on the iterator

Further control the iteration by adding a conditional to the end of the expression.

In [96]:
```
[val for val in range(20) if val % 3 > 0]
```

Out[96]:
```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The expression `(i % 3 > 0)` evaluate to `True` unless `val` is devisible by 3.

This also reads like English: "Construct a list of values for each value up to 20, but only if the value is not divisible by 3"

String manipulation

Python has some wonderful built-in string methods and formatting operations.

Such string manipulation patterns come up often in the context of data science work, and is one big perk of Python.

Strings can be defined using either single or double quotations:

In [97]:
```
x = 'a string'
y = "a string"
x == y
```

Out[97]:
```
True
```

Formatting strings: adjusting case

Python makes it quite easy to adjust the case of a string.

In [98]:
```
fox = "tHe qUICk bROWn fOx."
```

In [99]:
```
# convert into uppercase
fox.upper()
```

Out[99]:
```
'THE QUICK BROWN FOX.'
```

In [100]:
```
# convert into lowercase
fox.lower()
```

Out[100]:
```
'the quick brown fox.'
```

In [101]:
```
# Capitalise the first letter of each word
fox.title()
```

Out[101]:
```
'The Quick Brown Fox.'
```

In [102]:
```
# capitalise first letter of each sentence
fox.capitalize()
```

Out[102]:
```
'The quick brown fox.'
```

In [103]:
```
# or we can swap the cases
fox.swapcase()
```

Out[103]:
```
'ThE QuicK BrowN FoX.'
```

Formatting strings: adding and removing spaces

Sometimes we need to remove spaces (or other characters) from the begining or end of the string.

In [104]:
```
line = '             this is the content                 '
line.strip()
```

Out[104]:
```
'this is the content'
```

In [105]:
```
# remove just the space to the right
line.rstrip()
```

Out[105]:
```
'             this is the content'
```

In [106]:
```
# remove space to the left
line.lstrip()
```

Out[106]:
```
'this is the content                 '
```

In [107]:
```
# to remove other characters
num = '000000000000000435'
num.strip('0')
```

Out[107]:
```
'435'
```

We can add spaces or other characters

In [108]:
```
line = 'this is the content'

# center a given string within a given number of spaces:
line.center(30)
```

Out[108]:
```
'     this is the content      '
```

Similarly, `ljust()` and `rjust()` will left- or right-justrify the string.

In [109]:
```
# Fill with any character
'435'.rjust(10, '0')
```

Out[109]:
```
'0000000435'
```

Finding and replacing substrings

To find occurrences of a certain character in a string.

In [110]:
```
line = 'the quick brown fox jumped over a lazy dog'
line.find('fox')
```

Out[110]:
```
16
```

In [111]:
```
line.index('fox')
```

Out[111]:
```
16
```

The difference between the two methods is their behaviour when the search string is not found.

In [112]:
```
line.find('bear')
```

Out[112]:
```
-1
```

In [113]:
```
line.index('bear')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-113-adf599eb4c79> in <module>
----> 1 line.index('bear')

ValueError: substring not found
```

Splitting and partitioning strings

If you would like to find a substring and *then* split the string.

In [114]:
```
line.partition('fox')
```

Out[114]:
```
('the quick brown ', 'fox', ' jumped over a lazy dog')
```

In [115]:
```
line.split()
```

Out[115]:
```
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

Format strings

We can manipulate string *representations* of values of other types

In [116]:
```
pi = 3.14159
str(pi)
```

Out[116]:
```
'3.14159'
```

So we can do the following:

In [117]:
```
"The value of pi is " + str(pi)
```

Out[117]:
```
'The value of pi is 3.14159'
```

In [118]:
```
# a more flexible way is to use format
"The value of pi is {}".format(pi)
```

Out[118]:
```
'The value of pi is 3.14159'
```

The `format` method allows very powerful manipulation of the string.

In [119]:
```
# refer to the index of the argument:
"First Letter: {0}. Last letter: {1}.".format("A", "Z")
```

Out[119]:
```
'First Letter: A. Last letter: Z.'
```

In [120]:
```
# refer to the keyword argument:
"First: {first}. Last: {last}.".format(last='Z', first='A')
```

Out[120]:
```
'First: A. Last: Z.'
```

In [121]:
```
# include format codes that control how the value is converted to a string
"pi = {0:.3f}".format(pi)
```

Out[121]:
```
'pi = 3.142'
```

Modules and packages

Python standard libraries contains useful tools for a wide range of tasks.

There is also a huge range of third party tools and packages for more speialised functionalities.

We'll take a look at importing and installing modules.

Loading modules

Use the `import` statement to load built-in and third-party modules.

In [122]:
```
# explicit module import
import math
math.cos(math.pi)
```

Out[122]:
```
-1.0
```

In [123]:
```
# explicit module import by alias
import numpy as np
np.cos(np.pi)
```

Out[123]:
```
-1.0
```

In [124]:
```
# explicit import of module contents
from math import cos, pi
cos(pi)
```

Out[124]:
```
-1.0
```

In [125]:
```
# implicit import of module contents
from math import *
sin(pi) ** 2 + cos(pi) ** 2
```

Out[125]:
```
1.0
```

Warning: use `from ... import *` sparingly, if at all.

Python's standard library

Python has a large library of built-in modules, which you can find in Python's documentation (https://docs.python.org/3/library/)

A short list of some useful modules:

Module	Description
`os` and	Tools for interfacing with the operating system, including
`sys`	navigating file directory structures and executing shell commands
`math` and `cmath`	Mathematical functions and operations on real and complex numbers
`itertools`	Tools for constructing and interacting with iterators and generators
`functools`	Tools that assist with functional programming
`random`	Tools for generating pseudorandom numbers
`pickle`	Tools for object persistence: saving objects to and loading objects from disk
`json` and `csv`	Tools for reading JSON-formatted and CSV-formatted files

Third-party modules

Python has a huge library of third-party modules, you can import them just as the built-in module.

Two methods to install them on your system:

- `conda`
    - cross platform package and environment manager that installs and manages conda packages from the Anaconda repository as well as from the Anaconda Cloud
    - Conda keeps track of the dependencies between packages and platforms.
    - `conda install <package>`
- `pip`
    - Python Packaging Authority's recommended tool for installing packages from the Python Package Index (http://pypi.python.org/), PyPI
    - Pip installs Python software packaged as wheels or source distributions.
    - `pip install <package>`

Pip installs Python packages whereas conda installs packages which may contain software written in any language.

Pip and conda also differ in how dependency relationships within an environment are fulfilled. Pip doesn't check for dependencies of all packages are fulfilled simultaneously.

A preview of data science tools

We'll introduce and preview several of the more important packages for scientific computing and data science.

Install the packages through Anaconda or Miniconda

```
$ conda install numpy scipy pandas matplotlib scikit-learn
```

Numpy: numerical Python

An efficient way to store and manipulate multidimensional dense arrays in Python.

Provides an `ndarray` structure, which allows efficient storage and manipulation of vectors, matrices, and higher-dimensional datasets.

Provides a readable and efficient syntax for operating on this data.

In the simplest case, NumPy arrays looks like Python lists.

In [126]:
```
import numpy as np
x = np.arange(1, 10)
x
```

Out[126]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [127]:
```
# To square each element of the array
x ** 2
```

Out[127]:
```
array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Unlike Python lists, NumPy arrays can be multidimensional.

In [128]:
```
M = x.reshape((3, 3))
M
```

Out[128]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Pandas: labelled column-oriented data

A labelled interface to multidimensional data, in the form of a DataFrame object like that in R.

In [129]:
```
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
```

Out[129]:

	label	value
0	A	1
1	B	2
2	C	3
3	A	4
4	B	5
5	C	6

Manipulation of DataFrame object is very similar to those in R as well.

Matplotlib: MATLAB-style scientific visualisation

A powerful library for creating a large range of plots

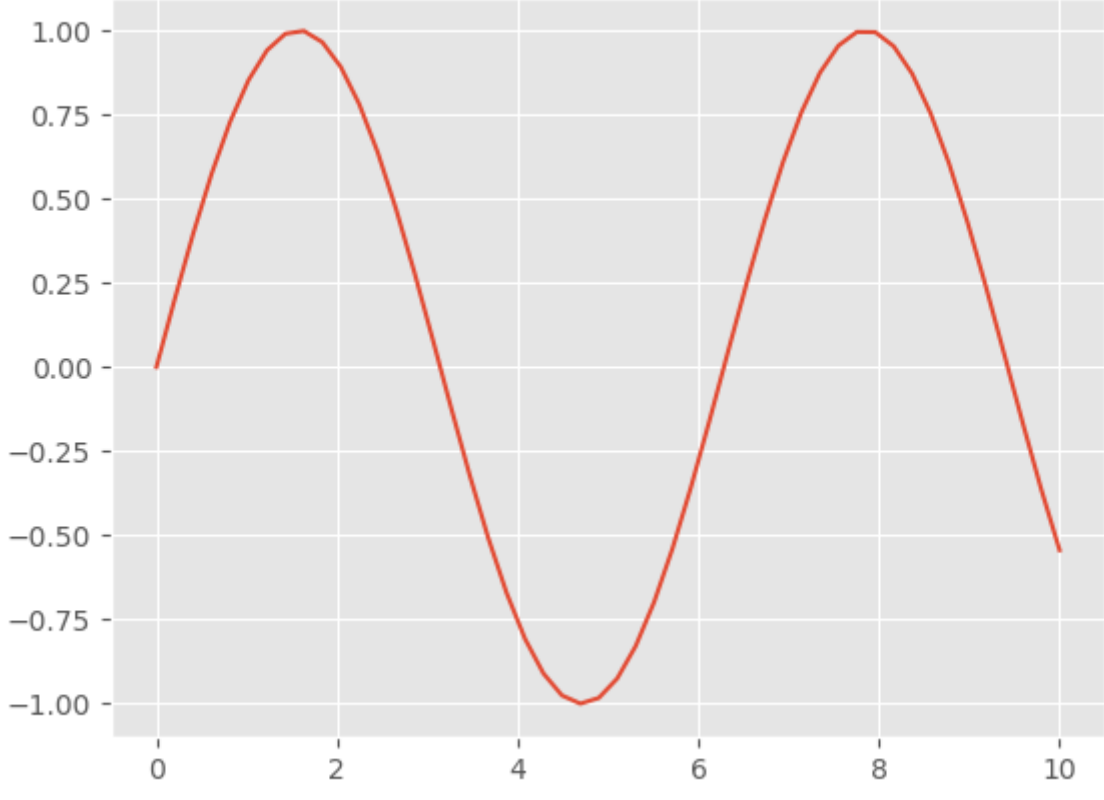
To use Matplotlib in Jupyter, enable the notebook mode:

In [130]:
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')    # make graphs in the style of ggplot
```

In [131]:
```
%matplotlib notebook
```

Create some data and plot the results:

In [132]:
```
x = np.linspace(0, 10)       # range of values from 0 to 10
y = np.sin(x)                 # sine of these values
plt.plot(x, y)                # plot as line
```



Out[132]:
```
[<matplotlib.lines.Line2D at 0x7fa96a6e3e10>]
```

SciPy: Scientific Python

A collection of scientific functionality that is built on Numpy.

The package has a set of sub-modules, each implementing some class of numerical algorithms.

sub-module	description
`scipy.fftpack`	Fast Fourier transforms
`scipy.integrate`	Numerical integration
`scipy.interpolate`	Numerical interpolation
`scipy.linalg`	Linear algebra routines
`scipy.optimize`	Numerical optimization of functions
`scipy.sparse`	Sparse matrix storage and linear algebra
`scipy.stats`	Statistical analysis routines

Scikit-learn: machine learning in Python

Provides a range of methods for data analysis and prediction

Allows a full data analysis pipeline:

- data pre-processing
- model selection
- dimensionality reduction
- common classification and regression methods
- clustering

Further Learning

There are a lot of resources for learning Python:

- *Fluent Python* by Luciano Ramalho
- *Dive into Python* by Mark Pilgrim
- *Learn Python the hard way* by Zed Shaw
- *Python essential reference* by David Beazley
- *The Python data science handbook* by Jake VanderPlas
- *Python for data analysis* by Wes McKinney