BCH2203 Python for Biochemistry - 3. Dictionaries and Functions

Ramses van Zon

Winter 2024

More composite data types

List-like data types

- **Sets:** are lists in which an element can only occur once.

  Elements in a set an unordered (so no indexing).

  Sets are defined with curly braces:

```
>>> s = {5, 4, 6, 5}
>>> print(s)
{4, 5, 6}
```

- **Tuples:** are lists that cannot be changed.

  Tuples are denoted with parentheses.

```
>>> t = (1,2,1)
>>> print(t)
(1, 2, 1)
```

- **Generators:** are lists generate their next element upon request.

```
>>> r = range(4)
>>> print(r)
range(0, 4)
>>> for x in r:
...     print(x)
...
0
1
2
3
```

- **Dictionaries:** are key-value hash tables.

Dictionaries

Key? Value? Hash? Table?

- In Python, a dictionaries (`dict` for short) is a look-up table.
- Think of the directory of a building:

- Suppose we need to allow look-up for 'Airbnb', 'SciNet', ...
- such that the result is the suite number in the MaRS building.

In Python:

```
>>> directory={'Airbnb': 10, 'SciNet': 1140}
>>> print(directory['SciNet'])
1140
```

Dictionaries

```
>>> directory={'Airbnb': 10, 'SciNet': 1140}
>>> print(directory['SciNet'])
1140
```

- A `dict` translates a 'key' to its associated 'value'.
- In the above example, the keys are 'Airbnb' and 'SciNet', and the values are 10 and 1140.
- You can give the key in square brackets to get the value out (a bit like a list).
- Keys must be unique, but can be integers, strings, ...
- Values can be anything.
- Other names for these structures:

  Associate Array, Map, Hash Map, Unordered Map

What can we do with these dicts?

- Look up values:

```
>>> directory['SciNet']
1140
```

- Get all the keys:

```
>>> directory.keys()
dict_keys(['SciNet', 'Airbnb'])
```

- Get all the values:

```
>>> directory.values()
dict_values([1140, 10])
```

- Add key-value pair:

```
>>> directory['TTC'] = 0
```

- Loop over all keys

```
>>> for k in directory:
...   print(k)
...
SciNet
Airbnb
TTC
```

- Loop over key-value pairs

```
>>> for k,v in directory.items():
...    print (k,v)
...
SciNet 1140
Airbnb 10
TTC 0
```

Interfacing with the Shell

Shebang comment

- Under Linux and MacOS, one often sees Python scripts that start with a "shebang" line:

```
#!/usr/bin/python
```

  or (much better)

```
#!/usr/bin/env python3
```

- Such a first line tells the operating system that this is a python script.
- Being identified as a python script, it can be executed from the terminal command line:

```
$ ./mypythonscript.py
```

  instead of

```
$ python mypythonscript.py
```

- For this to work, you might need to tell the OS additionally that this file may be executed:

```
$ chmod +x mypythonscript.py
```

Command-line arguments

What are Command-line arguments?

The terminal command line execution of python scripts, e.g.

```
$ python yearquery.py
```

can be augmented with arguments on the command line

```
$ python yearquery.py 1972 2000 2017
```

But this will not 'just work'. The script will need to be expecting command line arguments and deal with them.

Command-line arguments: `sys.argv`

- A script needs to expect command line arguments and deal with them, or those arguments will be ignored.
- The standard `sys` module contains a variable called `argv` which is a list of all command line arguments, called `argv`.
- To use it, import the `sys` module and use `sys.argv` as a list.

E.g.

```
#!/usr/bin/env python3
# file: sys_argv_example.py
import sys
print("You gave",len(sys.argv),"arguments")
for arg in sys.argv:
    print(arg)
```

With command line arguments, you can use a python script as a function within the shell, i.e., you could use it in shell scripts.

Functions

Functions

- Repeated code is bad: can make mistakes in twice as many lines of code.
- **Functions** are bits of reusable code.

  (Not the same as mathematical functions, mind you!)
- They are created with the `def` keyword.

Silly example

```
#!/usr/bin/env python3
# example_silly.py
def printfruit():
   print("apples and oranges")

printfruit()
printfruit()
```

```
$ python example_silly.py
apples and oranges
apples and oranges
```

Function arguments

- If functions did the same thing every time (like our example), they'd be of little use.
- The `print` function is an example of a less-trivial function.
- Depending on its *arguments*, the print function does something else.
- When we write our own functions, we can allow for one or more arguments too.
- The function definition must specify the names of the argument.

```
#!/usr/bin/env python3
# example_silly_argument.py
def printfruit(s)
    print (s, "apples and oranges")

printfruit("I like")
printfruit("I do not like")
```

```
$ python example_silly_argument.py
I like apples and oranges
I do not like apples and oranges
```

- `printfruit` takes one argument called `s`.
- Inside the function, `s` acts like a variable.
- After the function definition, we can use "`printfruit`" with different arguments.

More function arguments

- We can also have multiple function arguments.
- Simply list multiple names for function arguments, separated by commas.

```
#!/usr/bin/env python3
#example_fun_moreargs.py
def comparable(a,b):
    if a != b :
       print ('you cannot compare ', a, 'and', b)
    else:
       print (a, 'and', b, 'are comparable.')

comparable('apples','apples')
comparable('apples','oranges')
```

```
$ python example_fun_moreargs.py
apples and apples are comparable
you cannot compare apples and oranges
```

Function output

- So far our functions took arguments, i.e., input, but they produced output on the terminal.
- Functions are more useful if they produce output that can be put in a variable.
- To have your function produce such output, use the `return` statement.
- Whatever follows the return statement of a function becomes the function's return value. The function exits at the return statement.
- To use that return value to a variable, use the function call as if it's a variable.

Silly example

```
#!/usr/bin/env python3
# example_fun_output.py
def addone(x):
    return x+1

a=10
b=addone(a)
print(a,b)
```

```
$ python example_fun_output.py
10 11
$
```

More on functions

There's a lot more to be said about functions, the scope of variables, default values, variable number of arguments, keyword arguments, ...

Later.

Comments

Code is for humans

- So far, we have been writing **code for the computer.**
  (We tried to make the computer do something.)
- **Programs are meant to be read by humans and only incidentally for computers to execute.**
  (Harold Abelson, *Structure and Interpretation of Computer Programs*)
- How so?
  - Programmers spend more time reading someone else's code than writing their own.
  - There's no such this as a one-off script.
    (If you have saved a script as a file, it's no longer one-off, and you or someone else will eventually use it again and want to adapt it.)
- Readability, documentation, and maintainability very important.

How do you code for humans?

1. Write clear code.
2. Comment your code.
3. Document your code.

1. Write clear code: general principles

- **KISS: Keep It Simple, Stupid!**
  - Do not write code that is more complicated than necessary.
  - It will take too long for the next programmer of your future self to decode.
  - Your cleverest code will need someone cleverer than you to debug.
- **DRY: Don't repeat yourself.**

  Use functions to extract repeated code.
  $\Rightarrow$ Less code to figure out, less possible bugs, less code to maintain.
- **Separation of concerns**

  Each function should do only one thing, and do it well.

  This makes it easier to figure out, bugs become less complex, documentation becomes easier to write, and code can be reused in more situations (DRY).

1. Write clear code: style matters

A couple of code style guidelines can help too:

- Clear variable and function names
  `b` $\longrightarrow$ `calls_per_postal_code`
- One statement per line
- Use a consistent style
- Prefer small functions over long ones (as long as they perform a non-trivial task).
- Don't reinvent the wheel; use existing functions and packages.
- No cleverness.
- Use comments and documentation

2. Comment your code

- Comments start with #; anything after that is a comment.

```
amount_flour = 2.2 # amount in pounds
brand = 'Brand #1' # string contains '#'
```

- Keep comments succinct.
- Describe ***why*** your the code is doing something.

```
a_is_prime = all(a%i for i in range(2,a))
  # brute force determination of whether
  # a is prime (was easier to code than
  # a more sophisticated algorithm).
```

- Describe on a high level what parts of the code are doing.

```
# Get to user input an integer
while True:
  try:
    input_string = input("Enter an integer a=")
    a = int(input_string)
    break
  except:
    print("Not an integer; try again")
# Determine if integer a is prime
a_is_prime = all(a%i for i in range(2,a))
  # brute force determination of whether
  # a is prime (was easier to code than
  # a more sophisticated algorithm).
# Report back result to screen
print("a is prime?", a_is_prime)
```

3. Document your code

- You *would* also add comments describing what a function does, what parameters they take, and what they return.
- This would be the first line of defence in documenting that function.
- However, Python has a separate mechanism for such function documentation, called a docstring.
- Doc-strings are placed at the first line of the function.
- The docstring is returned by the `help` function.

```
>>> help(find_second)
```

```
>>> def find_second(searchin, forthis):
...   """Finds the 2nd occurance of a string.
...
...   Args:
...     searchin (str): string to search in.
...     forthis  (str): string to search for.
...
...   Returns:
...     int: The index in the search where the
...          second occurance starts.
...          -1 if there is no 2nd occurance.
...   """
...   # code would follow
...
```

The triple quotes are the python syntax for multi-line strings.

How to code for humans, continued

1. Write clear code
2. Comment your code
3. Document your code

Some schools of thought say that #2 and #3 are unnecessary if you write "self-documenting code".

Humbug!

Points #2 and #3 can be done simultaneously with Python **only up to a point**.

For more complex code, you will need to write files like `README`, `doc.txt`, `manual.pdf` ...

Writing modules

What are modules and packages?

- Modules are python files.
- Modules are meant to be imported into other python files or scripts.
- Convenient way to store functions that you might use in multiple projects.
- One or more modules can form a package.

Example

module file:

```
# file: yearprop.py
"""Deal with properties of calendar years."""
def is_leap_year(year):
  """Determines if a give year is a leap year.
  Argument year is the year to investigate.
  Returns True is year is a leap year, else
  False.
  """
  ...
```

usage in another script:

```
#!/usr/bin/env python3
# file: yearquery.py
"""Ask for years and say if they're leap years"""
import yearprop

# use the function yearprop.is_leap_year
year = int(input("Give a year"))
if yearprop.is_leap_year(year):
  print(year,"is a leap year")
else:
  print(year,"is not a leap year")
```

Module Example

module file:

```
# file: yearprop.py

"""Deal with properties of calendar years."""

def is_leap_year(year):
  """Determines if a give year is a leap year.
  Argument year is the year to investigate.
  Returns True is year is a leap year, else
  False.
   """
  if year%400 == 0:
    return True:
  elif year%100 == 0:
    return False
  elif year%4 ==0:
    return True
  else:
    return False
```

usage in another script:

```
#!/usr/bin/env python3
# file: yearquery.py
"""Ask for years and say if they're leap years"""
import yearprop
while True: # keep processing new input
  # keep asking input until we get an integer
  while True:
    try:
      year = int(input("Year (0 to stop)? "))
      break;
    except:
      print("That is not an integer.")
  # process input
  if yearprop.is_leap_year(year):
    print(year, "is a leap year!")
  else:
    print(year, "is not a leap year.")
  if year==0: break # stop processing input
print("Done")
```

Modules - Details

- We do not have to specify `.py` in the `import` statement.
- The file name without `.py` is the name of the module.
- The module file has to be in the same directory as the script, unless you install it (*later*)
- When putting functions in a module and importing that module, it gets put in the **namespace** of the modules. The name of the namespace is the name of the module.

  (in the example, the namespace was `yearprop`)