Doc ID: 6689693

```
class My_list(object):
    my_list = ['apples','milk','butter','orange juice']
    def __init__(self,snack='chocolate'):
        self.snack = snack
    def __str__(self):
        return 'My list: {}'.format(', '.join(self.my_list))

    def in_my_list(self,item):
        if item in self.my_list:
            return 'Got it!'
        else:
            return 'Nope!'
    def snack_check(self):
        return self.snack in self.my_list
```

```
#My favorite snack is chocolate... But in january I'm going to pretend it's oranges
jan = My_list('apples')
jan.snack_check()
```

```
#But in February, I'm back to the default
feb = My_list()
feb.snack_check()
```

About that object...

```
dir(object)
```

These are all the things you inherit by subclassing object.

```
class caps_list(My_list):
    def in_my_list(self,item):
        response = super(caps_list,self).in_my_list(item)
        return response.upper()
```

```
shouty = caps_list()
```

```
shouty.in_my_list('chocolate')
```

```
dir(caps_list)
```

You can also call the super class directly, like so:

Doc ID: 6689693

```
class caps_list(My_list):
    def in_my_list(self,item):
        # But you still have to pass self
        response = My_list.in_my_list(self,item)
        return response.upper()
```

```
shouty = caps_list()
shouty.in_my_list('chocolate')
```

Super actually assumes the correct things... Most of the time.

```
class caps_list(My_list):
    def in_my_list(self,item):
        response = super().in_my_list(item)
        return response.upper()
```

```
shouty = caps_list()
shouty.in_my_list('chocolate')
```

```
help(super)
```

# Lesson 08: Modules, Namespaces, and Packages

(b)(3)-P.L. 86-36

0

Updated over 2 years ago by [redacted] in COMP 3321

3 2 464 207

python

(U//~~FOUO~~) A lesson on Python modules, namespaces, and packages for COMP3321.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

## (U) Modules, Namespaces, and Packages

(U) We have already been using modules quite a bit -- every time we've run `import`, in fact. But what is a module, exactly?

## (U) Motivation

(U) When working in Jupyter, you don't have to worry about your code disappearing when you exit. You can save the notebook and share it with others. A Jupyter notebook kind of behaves like a python **script**: a text file containing Python source code. You can give that file to the python interpreter on the command line and execute all the code in the file (kind of like "Run All" in a Jupyter notebook):

```
$ python awesome.py
```

(U) There are a few significant limitations to sharing code in Jupyter notebooks, though:

1. what if you want to share with somebody who has python installed but not Jupyter?
2. what if you want to share *part* of the code with others (or reuse part of it yourself)?

3. what if you're writing a large, complex program?

(U) All of these *do* have native solutions in Jupyter:

1. convert the notebook to a script (File > Download as > Python)
2. copy-paste...?
3. make a big, messy notebook...?

(U) ...but they get unwieldy fast. This is where modules come in.

# (U) Modules

(U) At its most basic, a **module** in Python is really just another name for a script. It's just a file containing Python definitions and statements. The filename is the module's name followed by a `.py` extension. Typically, though, we don't run modules directly -- we *import* their definitions into our own code and use them there. Modules enable us to write *modular* code by organizing our program into logical units and putting those units in separate files. We can then share and reuse those files individually as parts of other programs.

# (U) Standard Modules

(U) Python ships with a library of standard modules, so you can get pretty far without writing your own. We've seen some of these modules already, and much of next week will be devoted to learning more about useful ones. They are documented in full detail in the Python Standard Library reference.

# (U) An awesome example

(U) To understand modules better, let's make our own. This will put some Python code in a file called `awesome.py` in the current directory.

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

a = Awesome("Everything")
print(a)
'''

with open('awesome.py', 'w') as f:
  f.write(contents)
```

(U) Now you can run `python awesome.py` on the command line as a Python script.

# (U) Using modules: `import`

(U) You can also import `awesome.py` here as a module:

```
import awesome
```

(U) Note that you leave out the file extension when you import it. Python knows to look for a file in your path called `awesome.py` .

(U) The first time you import the module, Python executes the code inside it. Any defined functions, classes, etc. will be available for use. But notice what happens when you try to import it again:

```
import awesome
```

(U) It's assumed that the other statements (e.g. variable assignments, print) are there to help *initialize* the module. That's why the module is only run once. If you try to import the same module twice, Python will not re-run the code -- it will refer back to the already-imported version. This is helpful when you import multiple modules that in turn import the same module.

(U) However, what if the module changed since you last imported it and you really want to do want to re-import it?

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

def cool(group):
    return "Everything is cool when you're part of {0}".format(group)

a = Awesome("Everything")
print(a)
'''

with open('awesome.py', 'w') as f:
  f.write(contents)
```

(U) You can bring in the new version with the help of the `importlib` module:

```
import importlib
importlib.reload(awesome)
```

## (U) Calling the module's code

(U) The main point of importing a module is so you can use its defined functions, classes, constants, etc. By default, we access things defined in the `awesome` module by prefixing them with the module's name.

```
print(awesome.Awesome("A Nobel prize"))
```

```
awesome.cool("a team")
```

```
print(awesome.a)
```

(U) What if we get tired of writing `awesome` all the time? We have a few options.

## (U) Using modules: `import ___ as ___`

(U) First, we can pick a nickname for the module:

```
import awesome as awe
```

```
print(awe.Awesome("A book of Greek antiquities"))
```

```
awe.cool("the Python developer community")
```

```
print(awe.a)
```

## (U) Using modules: `from ___ import ___`

(U) Second, we can import specific things from the `awesome` module into the current *namespace*:

```
from awesome import cool
```

```
cool("this class")
```

```
print(Awesome("A piece of string"))  # will this work?
```

```
print(a)  # will this work?
```

## (U) Get everything: `from ___ import *`

(U) Finally, if you really want to import *everything* from the module into the current namespace, you can do this:

```
from awesome import *  # BE CAREFUL
```

(U) Now you can re-run the cells above and get them to work.

(U) Why might you need to be careful with this method?

```
# what if you had defined this prior to import?
def cool():
    return "Something important is pretty cool"
```

```
cool()
```

## (U) Get one thing and rename: `from ___ import ___ as ___`

(U) You can use both `from` and `as` if you need to:

```
from awesome import cool as coolgroup
```

```
cool()
```

```
coolgroup("the A team")
```

## (U) Tidying up with `__main__`

(U) Remember how it printed something back when we ran `import awesome` ? We don't need that to print out every time we import the module. (And really aren't initializing anything important.) Fortunately, Python provides a way to distinguish between running a file as a script and importing it as a module by checking the special variable `__name__` . Let's change our module code again:

Doc ID: 6689693

```
contents = '''
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is awesome!!!".format(self)

def cool(group):
    return "Everything is cool when you're part of {0}".format(group)

if __name__ == '__main__':
    a = Awesome("Everything")
    print(a)
'''

with open('awesome.py', 'w') as f:
  f.write(contents)
```

(U) Now if you run the module as a script from the command line, it will make and print an example of the `Awesome` class. But if you import it as a module, it won't -- you will just get the class and function definition.

```
importlib.reload(awesome)
```

(U) The magic here is that `__name__` is the name of the current module. When you import a module, its `__name__` is the module name (e.g. `awesome`), like you would expect. But a running script (or notebook) also uses a special module at the top level called `__main__`:

```
__name__
```

(U) So when you run a module directly as a script (e.g. `python awesome.py`), its `__name__` is actually `__main__`, not the module name any longer.

(U) This is a common convention for writing a Python script: organize it so that its functions and classes can be imported cleanly, and put the "glue" code or default behavior you want when the script is run directly under the `__name__` check. Sometimes developers will also put the code in a function called `main()` and call that instead, like so:

```
def main():
    a = Awesome("Everything")
    print(a)

if __name__ == '__main__':
    main()
```

Doc ID: 6689693

# (U) Namespaces

(U) In Python, *namespaces* are what store the names of all variables, functions, classes, modules, etc. used in the program. A namespaces kind of behaves like a big dictionary that maps the name to the thing named.

(U) The two major namespaces are the *global* namespace and the *local* namespace. The global namespace is accessible from everywhere in the program. The local namespace will change depending on the current *scope* -- whether you are in a function, loop, class, module, etc. Besides local and global namespaces, each module has its own namespace.

# (U) Global namespace

(U) `dir()` with no arguments actually shows you the names in the global namespace.

```
dir()
```

(U) Another way to see this is with the `globals()` function, which returns a dictionary of not only the names but also their values.

```
sorted(globals().keys())
```

```
dir() == sorted(globals().keys())
```

```
globals()['awesome']
```

```
globals()['cool']
```

```
globals()['coolgroup']
```

# (U) Local namespace

(U) The local namespace can be accessed using `locals()`, which behaves just like `globals()`.

(U) Right now, the local namespace and the global namespace are the same. We're at the top level of our code, not inside a function or anything else.

```
globals() == locals()
```

(U) Let's take a look at it in a different scope.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

(U) You can access submodules by chaining them together with dot notation:

```
import sound.effects.reverse
```

(U) The other methods of importing work as well:

```
from sound.filters import karaoke
```

# (U) `__init__.py`

(U) What is this special `__init__.py` file?

- (U) Its presence is required to tell Python that the directory is a package
- (U) It can be empty, as long as it's there
- (U) It's typically used to initialize the package (as the name implies)

(U) `__init__.py` can contain any code, but it's best to keep it short and focused on just what's needed to initialize and manage the package. For example:

- (U) setting the `__all__` variable to tell Python what modules to include when someone runs `from package import *`
- (U) automatically import some of the submodules so that when someone runs `import package`, then they can run `package.function` rather than `package.submodule.function`

# (U) Installing packages

(U) Packages are actually the common way to share and distribute modules. A package can contain a single module -- there is no requirement for it to hold multiple modules. If you're wanting to work with a Python module that is not in the standard library (i.e. not installed with Python by default), then you will probably need to install the package that contains it. Python developers don't usually share or install individual module files.

## (U) pip and PyPI

(U) On the command line, the standard tool for installing a package is `pip`, Python's package manager. (`pip` ships with Python by default nowadays, but if you're using an older version, you may have to install it yourself.) To use `pip`, you need to configure it to point at a package repository. On the outside, the big repository everyone uses is called PyPI (a.k.a. the Cheese Shop).

## (U//~~FOUO~~) REPOMAN and nsa-pip

(U//~~FOUO~~) REPOMAN also imports and hosts a mirror of PyPI on the high side. Additionally, there is a nsa-pip server that connects to both REPOMAN's PyPI mirror and a variety of internal NSA-developed packages hosted on GitLab.

- (U//~~FOUO~~) List of internal NSA packages
- (U//~~FOUO~~) Links to some NSA package docs

## (U) ipydeps & pypki2

(U//~~FOUO~~) If you are working in a Jupyter notebook, it can be awkward trying to install packages from the command line with `pip` and then use them. Instead, `ipydeps` is a module that allows you to install packages directly from the notebook. It also uses the `pypki2` module behind the scenes to handle HTTPS connections that need your PKI certificates.

```
import ipydeps
ipydeps.pip('prettytable')
```

(b)(3)-P.L. 86-36

(U//~~FOUO~~) Another thing that `ipydeps` does behind the scenes is try to install operating system (non-Python) dependencies that the package needs in order to install and run correctly. That is manually configured by the Jupyter team here at NSA. If you run into trouble installing a package with `ipydeps` in Jupyter on LABBENCH, contact [redacted] and provide the name of the package you are trying to install and the errors you are seeing.

# Modules and Packages

(b)(3)-P.L. 86-36

0

Updated almost 3 years ago by [redacted] in COMP 3321

3 307 60

fcs6 access gitlab python

(U) Lesson 08: Modules and Packages

Recommendations

## (U) I see you like Python, so I put Python in your Python

(U) We've seen how to write scripts; now we want to reuse the good parts. We've already used the `import` command, which lets us piggyback on the work of others–either through Python's extensive standard library or through additional, separately-installed packages. It can also be used to proactively leverage the long tail of our own personal production. In this lesson, we cover, in much greater depth, the mechanics and principles of writing and distributing modules and packages. Suppose you have a script named `my_funcs.py` in your current directory. Then the following works just fine:

```
import my_funcs
```

```
import my_funcs as m
```

```
import importlib
importlib.reload(m)
```

```
from my_funcs import string_appender
```

```
from my_funcs import * # BE CAREFUL
```

(U) If you change the source file `my_funcs.py` in between `import` commands, you will have different versions of the functions imported. So what's going on?

## (U) Namespaces

(U) When you `import` a **module** (what we used to call merely a *script*), Python executes it as if from the command line, then places variables and functions inside a **namespace** defined by the script name (or using the optional `as` keyword). When you `from <module> import <name>`, the variables are imported into your current namespace. Think of a namespace as a super-variable that contains references to lots of other variables, or as a super-class that can contain data, functions, and classes.

(U) After import, a module is dynamic like any Python object; for example, the `reload` function takes a module as an argument, and you can add data and methods to the module after you've imported it (but they won't persist beyond the lifetime of your script or session).

```
import my_funcs as m
```

```
def silly_func(x):
    return "Silly {}!".format(x)
```

```
m.silly_func = silly_func
```

```
m.silly_func("Mark")
```

```
Silly Mark!
```

(U) In contrast, the `from <module> import <function>` command adds the function to the current namespace.

# (U) Preventing Excess Output: The Magic of `__main__`

(U) Suppose you have a script that does something awesome, called `awesome.py`:

```
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is AWESOME.".format(self)

a = Awesome("BASE Jumping")
print(a)
```

P.L. 86-36

(U) This can be executed from the command line or imported:

```
(VENV)[redacted]$ python awesome.py
BASE Jumping is AWESOME
(VENV)[redacted]$ python
```

```
import awesome
```

```
BASE Jumping is AWESOME.
```

```
a
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

```
awesome.a
```

```
<awesome.Awesome object at 0x7fa222a8b410>
```

```
print(awesome.a)
```

```
BASE Jumping is AWESOME.
```

(U) You don't want that print statement to execute every time you import it. Of equal importance, `awesome.a` is probably extraneous within an import. Let's fix it to get rid of those when you import the module, but keep them when you execute the script.

```
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is AWESOME.".format(self)

if __name__ == '__main__':
    a = Awesome("BASE Jumping")
        print(a)
```

(U) We can do even better. There are some situations, e.g. profiling or testing, where we would want to import the module, then look at what would happen if we run it as a script. To enable that, move the main functionality into a function called `main()` :

```
class Awesome(object):
    def __init__(self, awesome_thing):
        self.thing = awesome_thing
    def __str__(self):
        return "{0.thing} is AWESOME.".format(self)

def main():
    a = Awesome("BASE Jumping")
    print(a)

if __name__ == '__main__':
    main()
```

# (U) From Modules to Packages

(U) A single Python **module** corresponds to a **file**. It's not hard to imagine a situation where you have several related modules that you want to group together in the equivalent of a **folder**; the Python term for this concept is a **package**. We make a package by

- creating a folder
- putting scripts/modules inside it
- adding some Python Magic (which obviously will involve `__` in some way, shape, or form)

(U) For example, we'll put `awesome.py` in a package called feelings–later on, we'll add `terrible.py` and `totally_rad.py` The directory structure is:

```
feelings/
├── awesome.py
├── __init__.py
└── __main__.py
```

(U) The `__init__.py` file is **REQUIRED**; without it, Python won't identify this folder as a package. However, `__main__.py` is optional but nice; if you have it, you can type `python feelings` and the contents of `__main__.py` will be executed as a script. (NB: Now you can postulate on what `if __name__ == '__main__':` is really doing.

(U) The `__init__.py` file **can contain commands**. Much like the `__init__()` function of a `class` , the `__init__.py` is executed immediately after importing the package. One common use is to expose modules as package attributes; all this takes is `import <module_name>` in the package's `__init__.py` file.

# (U) Onward to the Whole World

Pretty soon, you'll want to share the **feelings** packages with a wider audience. There are thousands of people who want to do `Awesome` stuff, but don't have the time to make their own version, which wouldn't be as good as yours anyway, so they're counting on you to provide this package in a convenient, easy-to-install manner.

# (U) Shareable Packages

(U) The **_setuptools** package (which is built on **distutils**), used in conjunction with virtual environments and publicly accessible repositories in revision control systems make sharing your work as easy as `pip install` ing a package from PyPI. You are using a revision control system, aren't you? This lesson assumes that you use **git** and push your repositories to GitLab

(U) To make the **feelings** package available to the whole world, it should be placed at the root of a git repository, alongside a setup script called `setup.py` , i.e.

```
feelings_repo
├── feelings/
│   ├── awesome.py
│   ├── __init__.py
│   └── __main__.py
└── setup.py
```

(U) The `setup.py` script imports from one of two packages that handle management and installation of other packages. We'll use **setuptools** in this example, because it is more powerful and installed by default in virtual environments. In simple cases like this one, the built-in **distutils** module is more than adequate. and functionally identical.

(U) The script calls a single function, `setup` , and takes metadata about the package, including the name and version number of the package, the name and email of the developer responsible for the package, and a list of packages (or modules). It looks like this:

```
from setuptools import setup

setup(name="pyTest",
      version='0.0.1',
      description="The simplest Python Package imaginable",
      author=[redacted]
      author_email="[redacted]",
      packages=['feelings'],
      )
```

P.L. 86-36

(U) To use **distutils** instead of **setuptools**, change the first line to read `from distutils.core import setup`. Two powerful advantages of **setuptools** over **distutils** are:

- Dependency management, so that external packages available in PyPI will be installed automatically, and
- Automatic creation of *entry point* shell scripts that hook into specified functions in your code.

P.L. 86-36

# (U) Sharing Packages

(U) We have bigger fish to fry–we want to get the `Awesome`-ness out into the world, and we're almost there. Once the changes have been commited and pushed to GitLab, we can share them with one simple pip command. Inside of a virtual environment, anyone with access to GitLab can execute

```
$ pip install -e git+git@gitlab.coi.nsa.ic.gov:[redacted]/feelings.git#egg=feelings
```

(U) The `-e` flag installs the repository as *editable*, a.k.a. in *developer mode*. This means that the full git repository is cloned inside the virtual environment's `src` folder and can be modified or updated in place (e.g. using `git clone`) without requiring reinstallation. The `#egg=feelings` is necessary for `pip install` to work, and must be added manually; it is neither required nor even used by GitLab.

(U) Once your user has `pip install`-ed your package, that's it! She can now do awesome stuff, like

```
from feelings import awesome

a = awesome.Awesome("Dostoyevsky")

print(a)
```

```
Dostoyevsky is AWESOME.
```

(U) Even better, it only takes little more work for her to include your package as a dependency in her packages and applications!

# Lesson 09: Exceptions, Profiling, and Testing

0

(b)(3)-P.L. 86-36

Updated 8 months ago by [redacted] in COMP 3321

3 539 281

python fcs6

(U) Exception handling and code testing and profiling in Python.

Recommendations

UNCLASSIFIED

## (U) Introduction

(U) Attention to exception handling, profiling, and testing distinguishes professional developers writing high-quality code from amateurs that hack around just enough to get the job done. Each topic warrants many hours of discussion on its own, but Python makes it possible to start learning and using these principles with minimal effort. This section covers basic ideas to get you interested and see the usefulness of these ideas and modules. Let's begin...by making some errors.

## (U) Exceptions

(U) Python is very flexible and will try its absolute best to do whatever you ask it to, but sometimes you can just confuse it way too much. The first type of **error** is the **syntax error**. By this point in the course, we've all seen more than enough of these! They happens when Python cannot parse what you typed.

```
for i in range(10)
```

Doc ID: 6689693

```
def altered_cool():
    print(awesome.Awesome('Artisanal vinegar'))  # still there?
    print(coolgroup('the intelligentsia'))
    cool = 'hipster'
    lumberjack = True
    print(sorted(locals().keys()))
    print(locals()['cool'])
```

```
altered_cool()
```

```
'lumberjack' in globals()
```

```
globals()['cool']
```

```
globals() == locals()
```

# (U) Module namespaces

(U) Finally, each module also has its own *module* namespace. You can inspect them using the module's special `__dict__` method.

```
sorted(awesome.__dict__.keys())
```

```
# guess what?
dir(awesome) == sorted(awesome.__dict__.keys())
```

```
awesome.__dict__['cool']  # can also print this to get the memory location
```

```
# didn't we just see that here?
globals()['coolgroup']
```

```
dir(awe)
```

```
awe.__dict__['cool']
```

```
awe == awesome
```

```
id(awe) == id(awesome)
```

# (U) Modifying module namespaces

(U) You can add to module namespaces on the fly. Keep in mind, though, that this will only last until the program exits, and the actual module file will be unchanged.

```
def more_awesome():
  return "They're awesome!"

awe.exclaim = more_awesome
```

```
awe.exclaim()
```

```
'exclaim' in dir(awe)
```

```
'exclaim' in dir(awesome)
```

# (U) Packages

(U) What if you want to organize your code into multiple modules? Since a module is a file, the natural thing to do is to gather all your related modules into a single directory or folder. And, indeed, a Python **package** is just that: a directory that contains modules, a special `__init__.py` file, and sometimes more packages or other helper files.

Doc ID: 6689693

```
  File "<ipython-input-1-6f7914dd2e9a>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
```

(U) Python could not parse what we were trying to do here (because we forgot our colon). It did, however, let us know where things stopped making sense. Note the printed line with an tiny arrow ( ^ ) pointing to where Python thinks there is an issue.

(U) The statement `SyntaxError: invalid syntax` is an example of a special exception called a **SyntaxError**. It is fairly easy to see what happened here, and there is not much to do besides fixing your typo. Other **exceptions** can be much more interesting.

(U) There are many types of exceptions:

```
import builtins
```

```
# This will display a lot of output.
# To make it scrollable, select this cell and choose
# Cell > Current Output > Toggle Scrolling
help(builtins)
```

```
# Python 2 used to have this info in the `exceptions` module
# Python 3 moved it into `builtins` for consistency
# So for python 2, try this instead:
import exceptions
dir(exceptions)
```

(U) I bet we can make some of these happen. In fact, you probably already have recently.

```
1/0
```

```
def f():
    1/0
```

```
f()
```

```
1/'0'
```

```
import chris
```

```
file = open('data','w')
```

```
file.read()
```

# (U) Exception Handling

(U) When exceptions might occur, the best course of action to is to **handle** them and do something more useful than exit and print something to the screen. In fact, sometimes exceptions can be very useful tools (e.g. `KeyboardInterrupt` ). In Python, we handle exeptions with the `try` and `except` commands.

(U) Here is how it works:

1. (U) Everything between the `try` and `except` commands is executed.
2. (U) If that produces no exception, the `except` block is skipped and the program continues.
3. (U) If an exception occurs, the rest of the `try` block is skipped.
4. (U) If the type of exception is named after the `except` keyword, the code after the except command is executed.
5. (U) Otherwise, the execution stops and you have an **unhandled exception**.

(U) Everything makes more sense with an example:

```
def f(x):
    try:
        print ("I am going to convert the input to an integer")
        print (int(x))
    except ValueError:
        print ("Sorry, I was not able to convert that.")
```

```
f(2)
```

```
f('2')
```

```
f('two')
```

(U) You can add multiple Exception types to the except command:

```
... except (TypeError, ValueError):
```

(U) The keyword `as` lets us grab the message from the error:

```
def be_careful(a, b):
    try:
        print(float(a)/float(b))
    except (ValueError, TypeError, ZeroDivisionError) as detail:
        print("Handled Exception: ", detail)
    except:
        print("Unexpected error!")
    finally:
        print("THIS WILL ALWAYS RUN!")
```

```
be_careful(1,0)
```

```
be_careful(1,[1,2])
```

```
be_careful(1,'two')
```

```
be_careful(16**400,1)
```

```
float(16**400)
```

(U) We've also added the `finally` command. It will always be executed, regardless of whether there was an exception or not, so it should be used as a place to clean up anything left over from the `try` and `except` clauses, e.g. closing files that might still be open.

# (U) Raising Exceptions

(U) Sometimes, you will want to cause an exception and let someone else handle it. This can be done with the `raise` command.

```
raise TypeError('You submitted the wrong type')
```

(U) If no built-in exception is suitable for what you want to raise, defining a new type of exception is as easy as creating a new class that inherits from the `Exception` type.

```
class MyPersonalError(Exception):
    pass
```

```
raise MyPersonalError("I am mighty. Hear my roar!")
```

```
def locater (myLocation):
    if (myLocation<0):
        raise MyPersonalError("I am mighty. Hear my roar!")
    print(myLocation)
```

```
locater(-1)
```

(U) When catching an exception and raising a different one, both exceptions will be raised (as of Python 3.3).

```
class MyException(Exception):
    pass

try:
    int("abc")
except ValueError:
    raise MyException("You can't convert text to an integer!")
```

(U) You can override this by adding the syntax `from None` to the end of your `raise` statement.

```
class MyException(Exception):
    pass

try:
    int("abc")
except ValueError:
    raise MyException("You can't convert text to an integer!") from None
```

# (U) Testing

(U) There are two built-in modules that are pretty useful for testing your code. This also allows code to be tested each time it is imported so that a user on another machine would notice if certain methods did not do what they were intended to ahead of time.

# (U) The `doctest` Module

(U) The **doctest** module allows for testing of code and value assertions in the documentation of the code itself. It also works with exceptions; you just copy and paste the appropriate `Traceback` that is expected (just the first line and the actual exception string are needed). You may incorporate `doctest` into a module or script. See the official Python documentation for details.

Doc ID: 6689693

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
```

```
            raise ValueError("n must be exact integer")
        if n+1 == n:  # catch a value like 1e300
            raise OverflowError("n too large")
        result = 1
        factor = 2
        while factor <= n:
            result *= factor
            factor += 1
        return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

(U) This lesson can be tricky to understand from the notebook. It will make the most sense if you copy and paste the above code into a file named `factorial.py`, then from the terminal run:

```
python factorial.py -v
```

Note that you don't have to include the doctest lines in your code. If you remove them, the following should work:

```
python -m doctest -v factorial.py
```

# (U) The `unittest` Module

(U) The **unittest** module is much more stuctured, allowing for the developer to create a class of tests that are run and analyzed flexibly. To create a unit test for a module or script:

- `import unittest`,
- create a test class as a subclass of the `unittest.TestCase` type,
- add tests as methods of this class, making sure that the **name of each test function begins with the word 'test'**, and
- add `unittest.main()` to your main loop to run the tests.

```
import unittest
# ... other imports, script code, etc. ...
class FactorialTests(unittest.TestCase):
    def testSingleValue(self):
        self.assertEqual(factorial(5), 120)

    def testMultipleValues(self):
      ' self.assertRaises(TypeError, factorial, [1,2,3,4])

    def testBoolean(self):
        self.assertTrue(factorial(5) == 120)

def main():
    """ Main function for this script """
    unittest.main()  # Check the documentation for more verbosity levels, etc.
    # ... rest of main function ...

if __name__ == "__main__":
    main()
```

```
import unittest
dir(unittest.TestCase)
```

# (U) Profiling

(U) There are many profiling modules, but we will demonstrate the **cProfile** module from the standard library. To use it interactively, first import the module, then call it with a single argument, which must be a string that could be executed if it was typed into the interpreter. Frequently, this will be a previously-defined function.

```
import cProfile
```

Doc ID: 6689693

```
def long(upper_limit=100000):
    for x in range(upper_limit):
        pass
def short():
    pass
def outer(upper_limit=100000):
    short()
    short()
    long()
```

```
cProfile.run('outer()')
```

```
cProfile.run('outer(10000000)')
```

(U) The output shows

```
ncalls: the number of calls,
tottime: the total time spent in the given function (and excluding time made in calls to sub-functions),
percall: the quotient of tottime divided by ncalls
cumtime: the total time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recu
rsive functions.
percall: the quotient of cumtime divided by primitive calls
filename:lineno(function): provides the respective data of each function
```

(U) The quick and easy way to profile a whole application is just to call the **cProfile** main function with your script as an additional argument:

```
$ python -m cProfile myscript.py
```

(U) Another useful built-in profiler is **timeit**. It's well suited for quick answers to questions like "Which is better between A and B?"

```
$ python -m timeit "'for i in range(100):' ' str(i)'
```

```
import timeit
```

```
timeit.timeit('"-".join(str(n) for n in range(100))',number=20000)
```

Doc ID: 6689693

```
mySetup = '''
def myfunc(upper_limit=100000):
    return range(upper_limit)
'''
timeit.timeit('myfunc()',number=1000,setup=mySetup)
```

Exercise 1: Write a custom error and raise it if RangeQuery is created with dates not in the correct format.

Exercise 2: Given the list of tuples: [("2016-12-01", "2016-12-06"),("2015-12-01", "2015-12-06"),("2016-2-01", "2016-2-06"),("01/03/2014", "02/03/2014"), ("2016-06-01", "2016-10-06")] write a loop to print a rangeQuery for each of the date ranges using "~~TS//SI//REL TO USA, FVEY~~", "Primary IP address of Zendian diplomat", "10.254.18.162" as your classification, justification and selector.

Inside the loop, write a try/except block to catch your custom error for incorrectly formated dates.

# Lesson 10: Iterators, Generators and Duck Typing

(b)(3)-P.L. 86-36

0

Updated 9 months ago by [redacted] in COMP 3321

3 556 273

fcs6 python

(U) Iterators, generators, sorting, and duck typing in Python.

Recommendations

UNCLASSIFIED

## (U) Introduction: List Comprehensions Revisited

(U) We begin by reviewing the fundamentals of lists and list comprehension.

```
melist = [ i for i in range(1, 100, 2) ]
for i in melist:  # how does the loop work?
   print(i)
```

(U) What happens when the list construction gets more complicated?

```
noprimes = [ j for i in range(2, 19) for j in range(i*2, 500, i) ]
```

```
primes = [ x for x in range(2, 500) if x not in noprimes ]
```

```
print(sorted(primes))
```

(U) Can we do this in one shot? Yes, but...

```
# nesting madness !
primes = [ x for x in range(2, 500) if x not in [ j for i in range(2, 19) for j in range(i*2, 500, i) ] ]
```

# (U) Iterators

(U) To create your own iterable objects, suitable for use in `for` loops and list comprehensions, all you need to do is implement the right special methods for the class. The `__iter__` method should return the iterable object itself (almost always `self`), and the `__next__` method defines the values of the iterator.

(U) Let's do an example, sticking with the theme previously introduced, of an iterator that returns numbers in order, except for multiples of the arguments used at construction time. We'll make sure that it terminates eventually by raising the `StopIteration` exception whenever it gets to `200`. (This is a great example of an *exception* in Python that is not uncommon: handling an event that is not unexpected, but requires termination; `for` loops and list comprehensions *expect* to get the `StopIteration` exception as a signal to stop processing.)

```
class NonFactorIterable(object):
    def __init__(self, *args):
        self.avoid_multiples = args
        self.x = 0
    def __next__(self):
        self.x += 1
        while True:
            if self.x > 200:
                raise StopIteration
            for y in self.avoid_multiples:
                if self.x % y == 0:
                    self.x += 1
                    break
            else:
                return self.x
    def __iter__(self):
        return self
```

```
silent_fizz_buzz = NonFactorIterable(3, 5)
```

```
[x for x in silent_fizz_buzz]
```

```
mostly_prime = NonFactorIterable(2, 3, 5, 7, 11, 13, 17, 19)
```

```
partial_sum = 0
```

```
for x in mostly_prime:
    partial_sum += x
```

```
partial_sum
```

```
mostly_prime = NonFactorIterable(2, 3, 5, 7, 11, 13, 17, 19)
print(sum(mostly_prime))
```

(U) It may seem strange that the `__iter__` method doesn't appear to do anything. This is because in some cases the **iterator** for an object should not be the same as the object itself. Covering such usage is beyond the scope of the course.

(U) There is another way of implementing a custom iterator: the `__getitem__` method. This allows you to use the square bracket `[]` notation for getting data out of the object. However, you still must remember to raise a `StopIteration` exception for it to work properly in for loops and list comprehensions.

# Another iterator example

In the below example, we create an iterator that returns the squares of numbers. Note that in the `__next__` method, all we're doing is iterating our counter ( `self.x` ) and returning the square of that counter number, as long as the counter is not greater than the pre-defined limit ( `self.limit` ). The `while` loop in the previous example was specific to that use-case; we don't actually need to implement any looping at all in `__next__` , as that's simply the method called for each iteration through a loop on our iterator.

Here we're also implementing the `__getitem__` method, which allows us to retrieve a value from the iterator at a certain index location. This one simply calls the iterator using `self.__next__` until it arrives at the desired index location, then returns that value.

Doc ID: 6689693

```
class Squares(object):

    def __init__(self, limit=200):
        self.limit = limit
        self.x = 0

    def __next__(self):
        self.x += 1
        if self.x > self.limit:
            raise StopIteration
        return (self.x-1)**2

    def __getitem__(self, idx):
        # initialize counter to 0
        self.x = 0
        if not isinstance(idx, int):
            raise Exception("Only integer index arguments are accepted!")
        while self.x < idx:
            self.__next__()
        return self.x**2

    def __iter__(self):
        return self
```

```
my_squares = Squares(limit=20)
```

```
[x for x in my_squares]
```

```
my_squares[5]
```

```
# since we set a limit of 20, we can't access an index location higher than that
my_squares[25]
```

# (U) Benefits of Custom Iterators

1. (U) Cleaner code
2. (U) Ability to work with infinite sequences
3. (U) Ability to use built-in functions like `sum` that work with iterables
4. (U) Possibility of saving memory (e.g. `range` )

Doc ID: 6689693

# (U) Generators

(U) Generators are iterators with a much lighter syntax. Very simple generators look just like list comprehensions, except they're surrounded with parentheses `()` instead of square brackets `[]`. More complicated generators are defined like functions, with the one difference being that they use the `yield` keyword instead of the `return` keyword. A generator maintains state in between times when it is called; execution resumes starting immediately after the `yield` statement and continues until the next `yield` is encountered.

```
y = (x*x for x in range(30))
print(y)  # hmm...
```

```
def xsquared():
    for i in range(30):
        yield i*i
```

```
def xsquared_inf():
    x = 0
    while True:
        yield x*x
        x += 1
```

```
squares = [x for x in xsquared()]
print(squares)
```

(U) Another example...days of the week!

```
def day_of_week():
    i = 0
    days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
    while True:
        yield days[i%7]
        i += 1
day_of_week()
```

```
import random
def snowday(prob=.01):
    r = random.random()
    if r < prob:
        return "snowday!"
    else:
        return "regular day."
```

Doc ID: 6689693

```
n = 0
for x in day_of_week():
    today = snowday()
    print(x + " is a " + today)
    n += 1
    if today == "snowday!":
        break
```

```
weekday = (day for day in day_of_week())
```

```
next(weekday)
```

# (U) Pipelining

(U) One powerful use of generators is to connect them together into a *pipeline*, where each generator is used by the next. Since Python evaluates generators "lazily," i.e. as needed, this can increase the speed and potentially allow steps to run concurrently. This is especially useful if one or two steps can take a long time (e.g. a database query). Without generators, the long-running steps will become a bottleneck for execution, but generators allow other steps to proceed while waiting for the long-running steps to finish.

```
import random

# Get the fractional part of a string representation of a float
def frac_part(v):
  v = str(v)
  i, f = v.split('.')
  return f
```

```
# traditional approach
results = []
for i in range(20):
    r = random.random() *100      # generate a random number
    r_str = str(r)                # convert it to a string
    r_frac = frac_part(r_str)     # get the fractional part
    r_out = float('0.' + r_frac)  # convert it back to a float
    results.append(r_out)

results
```

```
# generator pipeline
rand_gen = ( random.random() * 100 for i in range(20) )
str_gen  = ( str(r) for r in rand_gen )
frac_gen = ( frac_part(r) for r in str_gen )
out_gen  = ( float('0.'+r) for r in frac_gen )

results = list(out_gen)
results
```

# (U) Sorting

(U) In Python 2, anything iterable can be sorted, and Python will happily sort it for you, even if the data is of mixed types--by default, it uses the built-in `cmp` function, which almost always does something (except with complex numbers). However, the results may not be what you expect!

(U) In Python 3, iterable objects must have the `__lt__` (`lt` = less than) function explicitly defined in order to be sortable.

(U) The built-in function `sorted(x)` returns a new list with the data from `x` in sorted order. The `sort` method (for `list`s only) sorts a list in-place and returns `None`.

```
int_data = [10, 1, 5, 4, 2]
```

```
sorted(int_data)
```

```
int_data
```

```
int_data.sort()
```

```
int_data
```

(U) To specify how the sorting takes place, both `sorted` and `sort` take an optional argument called `key`. `key` specifies a *function* of one argument that is used to extract a comparison key from each list element (e.g. `key=str.lower`). The default value is None (compare the elements directly).

```
users = ['hAcker1', 'TheBoss', 'botman', 'turingTest']
```

```
sorted(users)
```

```
sorted(users, key=str.lower)
```

(U) The `__lt__` function takes two arguments: `self` and another object, normally of the same type.

Doc ID: 6689693

```
class comparableCmp(complex):
    def __lt__(self, other):
        return abs(self) < abs(other)
```

```
a = 3+4j
```

```
b = 5+12j
```

```
a < b
```

```
a1 = comparableCmp(a)
```

```
b1 = comparableCmp(b)
```

```
a1 < b1
```

```
c = [b1, a1]
```

```
sorted(c)
```

(U) Here's how it works:

1. the argument given to `key` must be a function that takes a single argument;
2. internally, `sorted` creates function calls `key(item)` on each item in the list and then
3. sorts the original list by using `__lt__` on the results of the `key(item)` function.

(U) Another way to do the comparison is to use `key`:

```
def magnitude_key(a):
    return (a*a.conjugate()).real
```

```
magnitude_key(3+4j)
```

```
sorted([5+3j, 1j, -2j, 35+0j], key=magnitude_key)
```

(U) In many cases, we must sort a list of dictionaries, lists, or even objects. We could define our own key function or even several key functions for different sorting methods:

```
list_to_sort = [{'lname':'Jones', 'fname':'Sally'}, {'lname':'Jones', 'fname':'Jerry'}, {'lname':'Smith', 'fname':'John'}, {'lr
```

```
def lname_sorter(list_item):
    return list_item['lname']
```

```
def fname_sorter(list_item):
    return list_item['fname']
```

```
def lname_then_fname_sorter(list_item):
    return (list_item['lname'], list_item['fname'])
```

```
sorted(list_to_sort, key=lname_sorter)
```

```
sorted(list_to_sort, key=fname_sorter)
```

```
sorted(list_to_sort, key=lname_then_fname_sorter)
```

(U) While it's good to know how this works, this pattern common enough that there is a method in the standard library `operator` package to do it even more concisely.

```
import operator
```

```
lname_sorter = operator.itemgetter('lname') # same as previous lname_sorter
```

(U) The application of the `itemgetter` method returns a *function* that is eqivalent to the `lname_sorter` function above. Even better, when passed multiple arguments, it returns a tuple containing those items in the given order. Moreover, we don't even need to give it a name first, it's fine to do this:

```
sorted(list_to_sort, key=operator.itemgetter('lname'))
```

```
sorted(list_to_sort, key=operator.itemgetter('lname', 'fname')) # same as using lname_then_fname_sorter
```

(U) To use `operator.itemgetter` with `list` s or `tuple` s, give it integer indices as arguments. The equivalent function for objects is `operator.attrgetter` .

(U) Since we know so much about Python now, it's not hard to figure out how simple `operator.itemgetter` actually is; the following function is essentially equivalent:

```
def itemgetter_clone(*args):
    def f(item):
        return tuple(item[x] for x in args)
    return f
```

(U) Obviously, `operator.itemgetter` and `itemgetter_clone` are not actually simple--it's just that most of the complexity is hidden inside the Python internals and arises out of the fundamental data model.

# (U) Duck Typing

(U) All the magic methods we've discussed are examples of the fundamental Python principle of **duck typing**: "If it walks like a duck and quacks like a duck, it must be a duck." Even though Python has `isinstance` and `type` methods, it's considered poor form to use them to validate input inside a function or method. If verification needs to take place, it should be restricted to verifying required behavior using `hasattr`. The benefit of this approach can be seen in the built-in `sum` function.

```
help(sum)
```

(U) **Any** sequence of numbers, regardless of whether it's a `list`, `tuple`, `set`, generator, or custom iterable, can be passed to `sum`.

(U) The following is a comparison of *bad* and *good* examples of how to write a `product` function:

```
def list_prod(to_multiply):
    if isinstance(to_multiply, list): # don't do this!
        accumulator = 1
        for i in to_multiply:
            accumulator *= i
        return accumulator
    else:
        raise TypeError("Argument to_multiply must be a list")
```

```
def generic_prod(to_multiply):
    if hasattr(to_multiply, '__iter__') or hasattr(to_multiply, '__getitem__'):
        accumulator = 1
        for i in to_multiply:
            accumulator *= i
        return accumulator
    else:
        raise TypeError("Argument to_multiply must be a sequence")
```

```
list_prod([1,2,3])
```

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

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

(U) Having given that example, testing for iterability is one of a few special cases where `isinstance` might be the right function to use, but not in the obvious way. The `collections` package provides **abstract base classes** which have the express purpose of helping to determine when an object implements a common interface.

(U) Finally, effective use of duck typing goes hand in hand with robust error handling, based on the principle that "it's easier to ask for forgiveness than permission."

# Exercises

1. Add a method to your 'RangedQuery' class to allow instances of the class to be sorted by 'start_date'.
2. Write an iterator class 'ReverseIter' that takes a list and iterates it from the reverse direction.
3. Write a generator which will iterate over every day in a year. For example, the first output would be 'Monday, January 1'.
4. Modify the generator from exercise 2 so the user can specify the year and initial day of the week.

Doc ID: 6689693

# Pipelining with Generators

(b)(3)-P.L. 86-36

0

Created over 3 years ago by [redacted]
Python3 thumbnail 135 20

fcs6 data generators looping transformation pipeline pipelining processing laundry

(U) Defining processing pipelines with generators in Python. It's simply awesome.

Recommendations

# Pipelining with Generators

Imagine you're doing your laundry. Think about the stages involved. Roughly speaking, the stages are sorting, washing, drying, and folding. The beauty though is that even though these stages are sequential, they can be performed in parallel. This is called **pipelining**.

Python generators make pipelining easy and can even clarify your code quite a bit. By breaking your processing into distinct stages, the Python interpreter can make better use of your computer's resources, and even break the stages out into separate threads behind the scenes. Memory is also conserved because values are automatically generated as needed, and discarded as soon as possible.

A prime example of this is processing results from a database query. Often, before we can use the results of a database query, we need to clean them up by running them through a series of changes or transformations. Pipelined generators are perfect for this.

```
from pprint import pprint
import random
```

# A Silly Example

Here we're going to take 200 randomly generated numbers and extract their fractional parts (the part after the decimal point). There are probably more efficient ways to do this, but we're doing to do it by splitting out the string into two parts. Here we have a function that simply returns the integer part and the fractional part of an input float as two strings in a tuple.

```
def split_float(v):
    '''
    Takes a float or string of a float
    and returns a tuple containing the
    integer part and the fractional part
    of the number, as strings, respectively.
    '''
    v = str(v)
    i, f = v.split('.')
    return (i, '0.'+f)
```

# The Pipeline

Here we have a pipeline of four generators, each feeding the one below it. We pprint out the final resulting list after all the stages have complete. See the comments after each line for further explanation.

```
rand_gen = ( random.random() * 100 for i in range(200) )  # generate 200 random floats between 0 and 100, one at a time
results = ( split_float(r) for r in rand_gen )  # call our split_float() function which will generate the correspoding tuples
results = ( r[1] for r in results )  # we only care about the fractional part, so only keep that part of the tuple
results = ( float(r) for r in results )  # convert our fractional value from a string back into a float

pprint(list(results))  # print the final results
```

# Why not a for-loop?

We could have put all the steps of our pipeline into a single for-loop, but we get a couple advantages by breaking the stages out into separate generators:

- There's some clarity gained by having distinct stages specified as a pipeline. People reading the code can clearly see the transforms.
- In a for-loop, Python simply computes the values sequentially; there's no chance for automatic optimization or multi-threading. By breaking the stages out, each stage can execute in parallel, just like your washer and dryer.

# Another (Pseudo-)Example

Here's a pseudo-example querying a database that returns JSON that we need to convert to lists.

Doc ID: 6689693

```
import json
results = ( json.loads(result) for result in db_cursor.execute(my_query) )
results = ( r['results'] for r in results )
results = ( [ r['name'], r['type'], r['count'], r['source'] ] for r in results )
```

# Filters

We can even filter our data in our generator pipeline.

```
results = ( r for r in results if r[2] > 0 )   # remove results with a count of zero
foo(results)  # do something else with your results
```

# Lesson 11: String Formatting



0

Updated 9 months ago by [redacted] in COMP 3321

3 1 547 251

python fcs6

(U) Lesson 11: String Formatting

Recommendations

UNCLASSIFIED

## (U) Intro to String Formatting

(U) String formatting is a very powerful way to display information to your users and yourself. We have used it through many of our examples, such as this:

```
'This is a formatted String {}'.format("--->hi I'm a formatted String argument<---")
```

(U) This is probably the easiest example to demonstrate. The empty curly brackets `{}` take the argument passed into `format`.

(U) Here's a more complicated example:

```
'{2} {1} and {0}'.format('Henry', 'Bill', 'Bob')
```

(U) Arguments can be positional, as illustrated above, or named like the example below. Order does matter, but names can help.

```
'{who} is really {what}!'.format(who='Tony', what='awesome')
```

(U) You can also format lists:

```
cities = ['Dallas', 'Baltimore', 'DC', 'Austin', 'New York']

'{0[4]} is a really big city.'.format(cities)
```

(U) And dictionaries:

```
lower_to_upper = {'a':'A', 'b':'B', 'c':'C'}

"This is a big letter {0[a]}".format(lower_to_upper)  # notice no quotes around a
```

```
"This is a big letter {lookup[a]}".format(lookup=lower_to_upper)  # can be named
```

```
for little, big in lower_to_upper.items():
    print('[-->{0:10} -- {1:10}<--]'.format(little, big))
```

(U) If you actually want to include curly brackets in your printed statement, use double brackets like this: `{{ }}`.

```
"{{0}} {0}".format('Where do I get printed?')
```

(U) You can also store the format string in a variable ahead of time and use it later:

```
the_way_i_want_it = '{0:>6} = {0:>#16b} = {0:#06x}'

for i in 1, 25, 458, 7890:
    print(the_way_i_want_it.format(i))
```

# (U) Format Field Names

(U) Here are some examples of field names you can use in curly brackets within a format string.

`{<field name>}`

- (U) 1 : the *second* positional argument
- (U) name : keyword argument
- (U) 0.var : attribute named var of the first positional argument
- (U) 3[0] : element 0 of the fourth positional argument
- (U) me_data[key] : element associated with the specific key string 'key' of me_data

# (U) Format Specification

Doc ID: 6689693

(U) When using a format specification, it follows the field name within the curly brackets, and its elements must be in a certain order. This is only for reference; for a full description, see the Python documentation on string formatting.

```
{<field name>:<format spec>}
```

1. (U) Padding and Alignment

- `>` : align right
- `<` : align left
- `=` : only for numeric types
- `^` : center

1. (U) Sign

- `-` : prefix negative numbers with a minus sign
- `+` : like `-` but also prefix positive numbers with a `+`
- `' '` : like `-` but also prefix positive numbers with a space

1. (U) Base Indicator (precede with a hash `#` like above)

- `0b` : binary
- `0o` : octal
- `0x` : hexadecimal

1. (U) Digit Separator

- `,` : use a comma to separate thousands

1. (U) Field Width

- leading `0` : pad with zeroes at the front

1. (U) Field Type (letter telling which type of value should be formatted)

- `s` : string (the default)
- `b` : binary
- `d` : decimal: base 10
- `o` : octal
- `x` : hex uses lower case letters
- `X` : hex uses upper case
- `n` : like `d`, use locale settings to determine decimal point and thousands seperator
- no code integer : like `d`
- `e` : exponential with small e
- `E` : exponential with big E
- `f` : fixed point, `nan` for not a number and `inf` for infinity
- `F` : same as `f` but uppercase `NAN` and `INF`

- `g` : general format
- `G` : like `G` but uppercase
- `n` : locale settings like `g`
- `%` : times 100, displays as `f` with a `%`
- no code decimal : like `g` , precision of twelve and always one spot after decimal point

1. (U) Variable Width

# (U) New in Python 3.6: f-strings

```
# Add 'f' before the string to create an f-string
# Expression added directly inside the `{}` brackets rather than after the format statement
x = 34
y = 2
f"34 * 2 = {x*y}"
```

```
my_name = 'Bob'
f"My name is {my_name}"
```

# (U) Examples

```
'{0:{1}.{2}f}'.format(9876.5432, 18, 3)
```

```
'{0:010.4f}'.format(-123.456)
```

```
'{0:+010.4f}'.format(-123.456)
```

```
for i in range(1, 6):
    print('{0:10.{1}f}'.format(123.456, i))
```

```
v = {'value':876.543, 'width':15, 'precision':5}

"{0[value]:{0[width]}.{0[precision]}}".format(v)
```

```
data = [('Steve', 59, 202), ('Samantha', 49, 156), ('Dave', 61, 135)]

for name, age, weight in data:
    print('{0:<12s} {1:4d} {2:4d}'.format(name, age, weight))
```

```python
# same as above but with f-strings

data = [('Steve', 59, 202), ('Samantha', 49, 156), ('Dave', 61, 135)]

for name, age, weight in data:
    print(f'{name:<12s} {age:4d} {weight:4d}')
```

# COMP3321 Day01 Homework - GroceryList



Updated almost 3 years ago by [redacted] in COMP 3321

3 233 36

python exercises

(U) Homework for Day01 of COMP3321. Task is to sort items into bins.

Recommendations

## (U) COMP3321 Day01 Homework GroceryList

Doc ID: 6689693

```
myGroceryList = ["apples", "bananas", "milk", "eggs", "bread",
                 "hamburgers", "hotdogs", "ketchup", "grapes",
                 "tilapia", "sweet potatoes", "cereal",
                 "paper plates", "napkins", "cookies",
                 "ice cream", "cherries", "shampoo"]

## Items by category
vegetables = ["sweet potatoes", "carrots", "broccoli", "spinach",
              "onions", "mushrooms", "peppers"]
fruit = ["bananas", "apples", "grapes", "plumbs", "cherries", "pineapple"]
cold_items = ["eggs", "milk", "orange juice", "cheese", "ice cream"]
proteins = ["turkey", "tilapia", "hamburgers", "hotdogs", "pork chops", "ham", "meatballs"]
boxed_items = ["pasta", "cereal", "oatmeal", "cookies", "ketchup", "bread"]
paper_products = ["toilet paper", "paper plates", "napkins", "paper towels"]
toiletry_items = ["toothbrush", "toothpaste", "deodorant", "shampoo", "soap"]

## My items by category
my_vegetables = []
my_fruit = []
my_cold_items = []
my_proteins = []
my_boxed_items = []
my_paper_products = []
my_toiletry_items = []
```

(U) Fill in your code below. Sort the items in myGroceryList by type into appropriate my_category lists using looping and decision making

```
print("My vegetable list: ", my_vegetables)
print("My fruit list: ", my_fruit)
print("My cold item list: ", my_cold_items)
print("My protein list: ", my_proteins)
print("My boxed item list: ", my_boxed_items)
print("My paper product list: ", my_paper_products)
print("My toiletry item list: ", my_toiletry_items)
```

(U)

# Dictionary and File Exercises



0

Updated over 2 years ago by [redacted] in COMP 3321

3 317 33

fcs6 comp3321 file dictionary python exercises

(U) Dictionary and file exercises for COMP3321.

Recommendations

## Lists and Dictionary Exercises

### Exercise 1 (Euler's multiples of 3 and 5 problem)

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

```
multiples_3 = [i for i in range(3,1000,3)]
multiples_5 = [i for i in range(5,1000,5)]
multiples = set((multiples_3 + multiples_5)) # set will remove duplicate numbers
sum(multiples) # add all the numbers together
```

you can also do this in one line:

```
sum([i for i in range(3,1000) if i % 3 == 0 or i% 5 == 0])
```

### Exercise 2

Write a function that takes a list as a parameter and returns a second list composed of any objects that appear more than once in the original list

- duplicates([1,2,3,6,7,3,4,5,6]) should return [3,6]
- what should duplicates(['cow','pig','goat','horse','pig']) return?

```
# you can use a dictionary to keep track of the number of times seen
def duplicates(x):
    dup={}
    for i in x:
        dup[i] = dup.get(i,0)+1
    result = []
    for i in dup.keys():
        if dup[i] > 1:
            result.append(i)
    return result
```

```
x = [1,2,3,6,7,3,4,5,6]
duplicates(x)
```

```
#you can also just use lists...
def duplicates2(x):
  dup = []
  for i in x:
    if x.count(i) > 1 and i not in dup:
      dup.append(i)
  return dup
```

```
y = ['cow','pig','goat','horse','pig']
duplicates2(y)
```

```
z = ['2016','2015','2014']
duplicates(z)
```

# Exercise 3

Write a function that takes a portion mark as input and returns the full classification

- convert_classification('U//FOUO') should return 'UNCLASSIFIED//FOR OFICIAL USE ONLY'
- convert_classification('S//REL TO USA, FVEY') should return 'SECRET//REL TO USA, FVEY'

Doc ID: 6689694

```
# just create a "lookup table" for potenial portion marks
full_classifications = {'U//FOUO':'UNCLASSIFIED//FOR OFFICIAL USE ONLY',
                        'C//REL TO USA, FVEY':'CONFIDENTIAL//REL TO USA, FVEY',
                        'S//REL TO USA, FVEY':'SECRET//REL TO USA, FVEY',
                        'S//SI//REL TO USA, FVEY': 'SECRET//SI//REL TO USA, FVEY',
                        'TS//REL TO USA, FVEY': 'TOP SECRET//REL TO USA, FVEY',
                        'TS//SI//REL TO USA, FVEY': 'TOP SECRET//SI//REL TO USA, FVEY'}
def convert_classification(x):
   return full_classifications.get(x,'UNKNOWN')  # look up the value for the portion mark
```

```
convert_classification('U//FOUO')
```

```
convert_classification('S//REL TO USA, FVEY')
```

```
convert_classification('C//SI')
```

# File Input/Output Exercises

These exercises build on concepts in Lesson 3 (Flow Control, e.g., for loops) and Lesson 4 (Container Data Type, e.g, dictionaries). You will use all these concepts together with reading and writing from files

# First, Get the Data

Copy the sonnet from https://urn.nsa.ic.gov/t/tx6qm and paste it into a new text file named sonnet.txt.

# Exercise 1

Write a function called file_capitalize() that takes an input file name and an output file name, then writes each word from the input file with only the first letter capitalized to the output file. Remove all punctuation except apostrophe.

```
file_capitalize('sonnet.txt', 'sonnet_caps.txt') => capitalized words written to sonnet_caps.txt
```

```
# use help('') to see what each of these string methods are doing
def capitalize(sentence):
    words = sentence.split()  # use split to split the string by spaces (i.e., words)
    new_words = [ word.strip().capitalize() for word in words ] # captialize each word
    return ' '.join(new_words) # create and return one string by combing words with ' '

def remove_punct(sentence):
    # since replace() method returns a new string, you can chain calls to the replace()
    #  method in order to remove all punctuation in one line of code
    return sentence.replace('.', '').replace(',', '').replace(':', '').replace(';', '')

def file_capitalize(infile_name, outfile_name):
    infile = open(infile_name, 'r') # open the input file
    outfile = open(outfile_name, 'w') # open the outpu file

    for line in infile: # loop through each line of input
        outfile.write(capitalize(remove_punct(line)) + '\n') # write the capitalized version to the output file

    infile.close() # finally, close the files
    outfile.close()
```

```
file_capitalize('sonnet.txt', 'sonnet_caps.txt')
```

# Exercise 2

Make a function called file_word_count() that takes a file name and returns a dictionary containing the counts for each word. Remove all punctuation except apostrophe. Lowercase all words.

```
file_word_count('sonnet.txt') => { 'it': 4, 'me': 1, ... }
```

```
def file_word_count(infile_name):
    word_counts = {}

    with open(infile_name, 'r') as infile:  # using 'with' so we don't have to close the file
        for line in infile: # loop over each line in the file
            words = remove_punct(line)  # we can use the remove_punct from exercise above
            words = words.split() # split the line into words

            for word in words: # loop over each word
                word = word.strip().lower()
                 # add one to the current count for the word (start at 0 if not there)
                word_counts[word] = word_counts.get(word, 0) + 1

    return word_counts # return the whole dictionary of word counts
```

```
counts = file_word_count('sonnet.txt')
counts
```

# Extra Credit

Write the counts dictionary to a file, one key:value per line.

```
def write_counts(outfile_name, counts):
    with open(outfile_name, 'w', encoding='utf-8') as outfile:
        # to loop over a dictionary, use the items() method
        # items() will return a 2-element tuple containing a key and a value
        # below we pull out the values from the tuple into their own variables, word and count
        for word, count in counts.items():
            outfile.write(word + ':' + str(count) + '\n') # write out in key:value format
```

```
write_counts('sonnet_counts.txt', counts)  # use the counts dictionary from Exercise 2 above
```

(U)

# Structured Data and Dates Exercise

0



Updated over 3 years ago by [redacted] in COMP 3321

3 164 13

comp3321 datetime json csv data fcs6 exercises

(U) COMP3321 exercise for working with structured data and dates.

Recommendations

## Structured Data and Dates Exercise

Save the Apple stock data from https://urn.nsa.ic.gov/t/0grli to aapl.csv.

Use DictReader to read the records. Take the daily stock data and compute the average adjusted close ("Adj Close") per week. Hint: Use .isocalendar() for your datetime object to get the week number.

For each week, print the year, month, and average adjusted close to two decimal places.

```
Year 2015, Week 23, Average Close 107.40
Year 2015, Week 22, Average Close 105.10
```

Doc ID: 6689694

```
from csv import DictReader
from datetime import datetime

def average(numbers):
    if len(numbers) == 0:
        return 0.0

    return sum(numbers) / float(len(numbers))

def get_year_week(record):
    dt = datetime.strptime(record['Date'], '%Y-%m-%d')
    return (dt.year, dt.isocalendar()[1])

def get_averages(data):
    avgs = {}

    for year_week, closes in data.items():
        avgs[year_week] = average(closes)

    return avgs

def weekly_summary(reader):
    weekly_data = {}

    for record in reader:
        year_week = get_year_week(record)

        if year_week not in weekly_data:
            weekly_data[year_week] = []

        weekly_data[year_week].append(float(record['Adj Close']))

    return get_averages(weekly_data)

def file_weekly_summary(infile_name):
    with open(infile_name, 'r') as infile:
        return weekly_summary(DictReader(infile))

def print_weekly_summary(weekly_data):
    for year_week in reversed(sorted(weekly_data.keys())):
```

```
        year = year_week[0]
        week = year_week[1]
        avg = weekly_data[year_week]
        print('Year {year}, Week {week}, Average Close {avg:.2f}'.format(year=year, week=week, avg=avg))
```

```
data = file_weekly_summary('aapl.csv')
print_weekly_summary(data)
```

# Extra

Use csv.DictWriter to write this weekly data out to a new CSV file.

```
from csv import DictWriter

def write_weekly_summary(weekly_data, outfile_name):
    headers = [ 'Year', 'Week', 'Avg' ]

    with open(outfile_name, 'w', newline='') as outfile:
        writer = DictWriter(outfile, headers )
        writer.writeheader()

        for year_week in reversed(sorted(weekly_data.keys())):
            rec = { 'Year': year_week[0], 'Week': year_week[1], 'Avg': weekly_data[year_week] }
            writer.writerow(rec)
```

```
data = file_weekly_summary('aapl.csv')
write_weekly_summary(data, 'aapl_summary.csv')
```

# Extra Extra

Use json.dumps() to write a JSON entry for each week on a new line.

```
import json

def write_json_weekly_summary(weekly_data, outfile_name):
    with open(outfile_name, 'w') as outfile:
        for year_week in reversed(sorted(weekly_data.keys())):
            rec = { 'year': year_week[0], 'week': year_week[1], 'avg': weekly_data[year_week] }
            outfile.write(json.dumps(rec) + '\n')
```

Doc ID: 6689694

```
data = file_weekly_summary('aapl.csv')
write_json_weekly_summary(data, 'aapl.json')
```

(U)

# Datetime Exercise Solutions



0

Created almost 3 years ago by [redacted] in COMP 3321

3 1 143 17

comp3321 exercises

(U) Solutions for the Datetime exercises

Recommendations

## (U) Datetime Exercises

(U) How long before Christmas?

```
import datetime, time

print(datetime.date(2017, 12, 25) - datetime.date.today())
```

***(U) Or, if you're counting the microseconds:***

```
print(datetime.datetime(2017, 12, 25) - datetime.datetime.today())
```

(U) How many seconds since you were born?

```
birthdate = datetime.datetime(1985, 1, 31)
time_since_birth = datetime.datetime.today() - birthdate
print('{:,}'.format(time_since_birth.total_seconds()))
```

(U) What is the average number of days between Easter and Christmas for the years 2000 - 2999?

Doc ID: 6689694

```
from dateutil.easter import easter
total = 0
span = range(2000, 3000)
for year in span:
    total += (datetime.date(year, 12, 25) - easter(year)).days

average = total / len(span)
print('{:6.4f}'.format(average))
```

(U) What day of the week does Christmas fall on this year?

```
datetime.date(2015, 12, 25).strftime('%A')
```

(U) You get a intercepted email with a POSIX timestamp of 1435074325. The email is from the leader of a Zendian extremist group and says that there will be an attack on the Zendian capitol in 14 hours. In Zendian local time, when will the attack occur? (Assume Zendia is in the same time zone as Kabul)

```
import pytz
utc_tz = pytz.timezone('Etc/UTC')
email_time_utc = datetime.datetime.fromtimestamp(1435074325, tz=utc_tz)
attack_time_utc = email_time_utc + datetime.timedelta(hours=14)
zendia_tz = pytz.timezone('Asia/Kabul')
attack_time_zendia = attack_time_utc.astimezone(zendia_tz)

print(email_time_utc)
print(attack_time_utc)
print(attack_time_zendia)
```

Doc ID: 6689694

(U)

# Object Oriented Programming and Exceptions Exercise

(b)(3)-P.L. 86-36

0

Created over 3 years ago by [redacted] in COMP 3321

3 2 125 15

fcs6 oop comp3321 exceptions objects classes exercises

(U) COMP3321 exercise for object oriented programming and exceptions.

Recommendations

# Object Oriented Programming and Exceptions Exercise

Make a class called Symbol that holds data for a stock symbol, with the following properties:

```
self.name
self.daily_data
```

It should also have the following functions:

```
def __init__(self, name, input_file)
def data_for_date(self, date_str)
```

**init**(self, name, input_file) should open the input file and read it with DictReader, putting each entry in self.daily_data, using the date strings as the keys. Make sure to open the daily data file within a try/except block in case the file does not exist. If the file does not exist, set self.daily_data to an empty dictionary.

data_for_date(self, date_str) should take a date string and return the dictionary containing that days' data. If there is no entry for that date, return an empty dictionary.

Doc ID: 6689694

# Tests

Make sure the following execute as specified in each comment. You can get the aapl.csv file from https://urn.nsa.ic.gov/t/0grli. The apple.csv file should not exist.

```
s1 = Symbol('AAPL', 'aapl.csv')
print(s1.data_for_date('2015-08-10'))  # should return a dictionary for that date
print(s1.data_for_date('2015-08-09'))  # should return an empty dictionary

s2 = Symbol('AAPL', 'apple.csv')       # should not raise an exception!
print(s2.data_for_date('2015-08-10'))  # should return an empty dictionary
print(s2.data_for_date('2015-08-09'))  # should return an empty dictionary
```

# Module: Collections and Itertools



Updated almost 2 years ago by [redacted] in COMP 3321

3 1 296 110

fcs6 python

(U) Module: Collections and Itertools

Recommendations

(U) Any programming language has to strike a balance between the number of basic elements it exposes, like control structures, data types, and so forth, and the utility of each one. For example, Python could do without `tuple` s entirely, and could replace the `dict` with a `list` of `list` s or even a single `list` where even-numbered indices contain *keys* and odd-numbered indices contain *values*. Often, there are situations that happen so commonly that they warrant inclusion, but inclusion in the **builtin** library is not quite justified. Such is the case with the **collections** and **itertools** modules. Many programs could be simplified with a `defaultdict` , and having one available with a single `from collection import defaultdict` is much better than reinventing the wheel every time it's needed.

## (U) Value Added Containers with **collections**

(U) Suppose we want to build an index for a poem, so that we can look up the lines where each word occurs. To do this, we plan to construct a dictionary with the words as keys, and a list of line numbers is the value. Using a regular `dict` , we'd probably do something like this:

```
poem = """mary had a little lamb
it's fleece was white as snow
and everywhere that mary went
the lamb was sure to go"""
```

```
index = {}
```

Doc ID: 6689695

```
for linenum, line in enumerate(poem.split('\n')):
    for word in line.split():
        if word in index:
            index[word].append(linenum)
        else:
            index[word] = [linenum]
```

(U) This code would be simpler without the inner `if ... else ...` clause. That's exactly what a `defaultdict` is for; it takes a function (often a `type`, which is called as a constructor without arguments) as its first argument, and calls that function to create a *default* value whenever the program tries to access a key that isn't currently in the dictionary. (It does this by overriding the `__missing__` method of `dict`.) In action, it looks like this:

```
from collections import defaultdict
```

```
index = defaultdict(list)
```

```
for linenum, line in enumerate(poem.split('\n')):
    for word in line.split():
        index[word].append(linenum)
```

(U) Although a `defaultdict` is almost exactly like a dictionary, there are some possible complications because it is possible to add keys to the dictionary unintentionally, such as when testing for membership. These complications can be mitigated with the `get` method and the `in` operator.

```
'sheep' in index # False
```

```
1 in index.get('sheep') # Error
```

```
'sheep' in index # still False
```

```
2 in index['sheep'] # still False, but...
```

```
'sheep' in index # previous statement accidentally added 'sheep'
```

(U) You can do crazy things like change the `default_factory` (it's just an attribute of the `defaultdict` object), but it's not commonly used:

```
import itertools
```

```
def constant_factory(value):
    return itertools.repeat(value).__next__
```

```
d = defaultdict(constant_factory('<missing>'))
```

```
d.update(name='John', action='ran')
```

Doc ID: 6689695

```
'{0[name]} {0[action]} to {0[object]}'.format(d)
```

```
d # "object" added to d
```

(U) A `Counter` is like a `defaultdict(int)` with additional features. If given a `list` or other iterable when constructed, it will create counts of all the unique elements it sees. It can also be constructed from a dictionary with numeric values. It has a custom implementation of `update` and some specialized methods, like `most_common` and `subtract`.

```
from collections import Counter
```

```
word_counts = Counter(poem.split())
```

```
word_counts.most_common(3)
```

```
word_counts.update('lamb lamb lamb stew'.split())
```

```
word_counts.most_common(3)
```

```
c = Counter(a=3, b=1)
```

```
d = Counter(a=1, b=2)
```

```
c + d
```

```
c - d          # Did you get the output you expected?
```

```
(c - d) + d
```

```
c & d
```

```
c | d
```

(U) An `OrderedDict` is a dictionary that remembers the order in which keys were originally inserted, which determines the order for its iteration. Aside from that, it has a `popitem` method that can pop from either the beginning or end of the ordering.

(U) `namedtuple` is used to create lightweight objects that are somewhat like tuples, in that they are immutable and attributes can be accessed with `[]` notation. As the name indicates, attributes are named, and can also be accessed with the `.` notation. It is most often used as an optimization, when speed or memory requirements dictate that a `dict` or custom object isn't good enough. Construction of a `namedtuple` is somewhat indirect, as `namedtuple` takes field specifications as strings and returns a `type`, which is then used to create the named tuples. named tuples can also enhance code readability.

```
from collections import namedtuple
```

```
Person = namedtuple('Person', 'name age gender')
bob = Person(name='Bob', age=30, gender='male')
print( '%s is a %d year-old %s' % bob )  # 2.x style string formatting
print( '{} is a {} year-old {}'.format(*bob) )
print( '%s is a %d year-old %s' % (bob.name, bob.age, bob.gender) )
print( '{} is a {} year-old {}'.format(bob.name, bob.age, bob.gender) )
bob[0]
bob['name']                 # TypeError
bob.name
print( '%(name)s is a %(age)d year-old %(gender)s' % bob )    # Doesn't work
print( '{name} is a {age} year-old {gender}'.format(*bob) )  # Doesn't work
print( '{0.name} is a {0.age} year-old {0.gender}'.format(bob) )  # Works!
```

(U) Finally, `deque` provides queue operations.

```
from collections import deque
d = deque('ghi')       # make a new deque with three items
d.append('j')          # add a new entry to the right side
d.appendleft('f')      # add a new entry to the left side
d.popleft()            # return and remove the leftmost item
d.rotate(1)            # right rotation
d.extendleft('abc')    # extendleft() reverses the input order
```

(U) The **collections** module also provides Abstract Base classes for common Python interfaces. Their purpose and use is currently beyond the scope of this course, but the documentation is reasonably good.

Doc ID: 6689695

# (U) Slicing and Dicing with **itertools**

Given one or more `list` s, `iterator` s, or other iterable objects, there are many ways to slice and dice the constituent elements.
The **itertools** module tries to expose building block methods to make this easy, but also tries to make sure that its methods are useful in a variety of situations, so the documentation contains a cookbook of common use cases. We only have time to cover a small subset of the **itertools** functionality. Methods from **itertools** usually return an iterator, which is great for use in loops and list comprehensions, but not so good for inspection; in the code blocks that follow, we often call `list` on these things to unwrap them.

(U) The `chain` method combines iterables into one super-iterable. The `groupby` method separates one iterator into groups of adjacent objects, possibly as determined by an optional argument–this can be tricky, especially because there's no look back to see if a new key has been encountered previously.

```
import itertools
```

```
list(itertools.chain(range(5),[5,6])) == [0,1,2,3,4,5,6]
```

```
size_groups = itertools.groupby([1,1,2,2,2,'p','p',3,4,3,3,2])
```

```
[(key, list(vals)) for key, vals in size_groups]
```

(U) A deeply nested for loop or list comprehension might be better served by some of the *combinatoric generators* like `product` , `permutations` , or `combinations` .

```
iter_product = itertools.product([1,2,3],['a','b','c'])
```

```
list(iter_product)
```

```
iter_combi = itertools.combinations("abcd",3)
```

```
list_combi = list(iter_combi)
list_combi
```

```
iter_permutations = itertools.permutations("abcd",3)
```

```
list(iter_permutations)
```

(U) `itertools` can also be used to create generators:

```
counter = itertools.count(0, 5)
```

```
next(counter)
```

```
print(list(next(counter) for c in range(6)))
```

(U) Be careful... What's going on here?!?

```
counter = itertools.count(0.2,0.1)
for c in counter:
    print(c)
    if c > 1.5:
        break
```

```
cycle = itertools.cycle('ABCDE')
```

```
for i in range(10):
    print(next(cycle))
```

```
repeat = itertools.repeat('again!')
```

```
for i in range(5):
    print(next(repeat))
```

```
repeat = itertools.repeat('again!', 3)
for i in range(5):
    print(next(repeat))
```

```
nums = range(10,0,-1)
my_zip = zip(nums, itertools.repeat('p'))
for thing in my_zip:
    print(thing)
```

Doc ID: 6689695

# Functional Programming

(b)(3)-P.L. 86-36

0

Created over 3 years ago by [redacted] in COMP 3321

3 2 193 38

fcs6 functional map python reduce

(U//~~FOUO~~) A short adaptation of [redacted] "A practical introduction to functional programming" in Python to supplement COMP 3321 materials. Also discusses lambdas.

Recommendations

UNCLASSIFIED

## (U) Introduction

(U) At a basic level, there are two fundamental programming styles or paradigms:

- *imperative* or *procedural* programming and
- *declarative* or *functional* programming.

(U) Imperative programming focuses on telling a computer how to change a program's *state*--its stored information--step by step. Most programmers start out learning and using this style. It's a natural outgrowth of the way the computer actually works. These instructions can be organized into functions/procedures (*procedural* programming) and objects (*object-oriented*programming), but those stylistic improvements remain imperative at heart.

(U) Declarative programming, on the other hand, focuses on expressing *what* the program should do, not necessarily *how* it should be done. *Functional programming* is the most common flavor of that. It treats a program as if it is made up of *mathematical*-style functions: for a given input `x`, running it through function `f` will always give you the same output `f(x)`, and `x` itself will remain unchanged afterwards. (Note that this is *not* necessarily the same as a procedural-style function, which may have access to global variables or other "inputs" and which may be able to modify those inputs directly.)