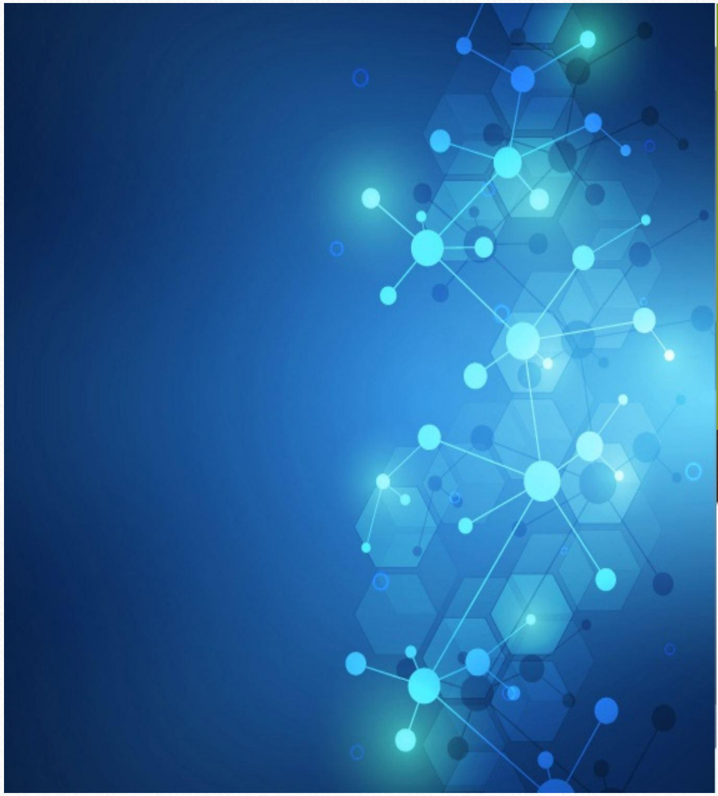
MSE 800

- Professional Software Engineering

week3



Outline:

- File Processing
- Venn Diagram
- Module
- Namespace
- More on Functions

File Processing

Types of file-processing tasks

- HUGE range, for example
  - Numerical / scientific data processing (e.g., rainfall data) Our Focus
  - Commercial data processing (e.g., files of account transactions)
  - Document processing (e.g., MS Word documents)
  - Programming language compilation (e.g., a Fortran program)
  - Image processing (e.g., green screening)
  - Internet data harvesting (e.g., web-crawling for email addresses)
  - ...

Steps in processing numerical data

1. Open the file
2. Extract the data from the file
   - May be as simple as splitting each line in a *.csv* file or as complex as parsing an XML file
3. Process the data
4. Output/display the results

Files as sequences

- Built-in function *open(path ,mode)* opens a file
  - *path* is a *filepath* string, for example, "H:/121/junk.txt"
    - Python allows forward-slashes instead of Windows' backslashes
    - If you want backslashes, "H:\\121\\junk.txt"
  - mode is "r" (default), "w" or "a" to read, write or append, respectively
- A file object, opened for reading, is a sequence of *lines*.

Files as lists of lines

```
data = open("junk.txt")  # Default is open for reading
for line in data:  # Processes file line by line
    print(line[0:-1])  # Print the line without its final \n char
data.close() # Do not need the file any more
```

```
data = open("junk.txt")
lines = data.readlines()  # Get a list of all the lines in the file
for line in lines:        # Processes file line by line
    print(line[0:-1])  # Print the line without its final \n char
data.close()
```

Files as lists of lines

```
data = open("junk.txt")
lines = data.readlines()  # Get a list of all the lines in the file
for line in lines:        # Processes file line by line
    print(line[0:-1])     # Print the line without its final \n char
data.close()
```

```
with open("junk.txt", "r") as data:
    lines = data.readlines()
    for line in lines:
        print(line[0:-1])
```

Read, Write, Append

```
with open("junk.txt", "a") as file:
    file.write("Appended line 1\n")
    file.write("Appended line 2\n")

with open("junk.txt", "w") as file:
    file.write("Hello, World\n")
    file.write("This is a test\n")
    file.write("Hello, Python\n")
```

Simple Data Processing Example

- File "mean_temperature.txt" of temperature measurements:
- Each line contains one measurement:

  month, day, hour, mean temperature
- Items in a line are separated by whitespace.
- What is the maximum temperature?

```
8   13    10 7.5
8   13    11 9.2
8   13    12 11.7
…
```

Code for Data Processing Example

```python
infile = open("mean_temperature.txt", "r")

mean_temps = []
for line in infile:
    data = line.split(' ')
    mean_temp = float(data[3])
    mean_temps.append(mean_temp)
print(max(mean_temps))
infile.close()
```

Extracting data

- Real data files usually have lots of unrelated info
  - Headers, footers, unrelated data, etc
  - For example, see next slide
    - o The result of querying for sunshine data at Christchurch from http://cliflo.niwa.co.nz
- Need an *algorithm* to extract just the required data
  - for example: month, day, sunshine from following slide

```
Name,Agent Number,Network Number,Latitude (dec.deg),Longitude
(dec.deg),Height (m),...
Christchurch Aero,4843,H32451,-43.493,172.537,37,G,N/A
Note: Position precision types are: "W" = based on whole
minutes, "T" = estimated to ... "G" = derived from gridref ,
"E" = error cases derived from gridref,
"H" = based on GPS readings (NZGD49), "D" = by definition i.e.
grid points.

Sunshine: Daily
Station,Date(NZST),Time(NZST),Amount(Hrs)
,Period(Hrs),Frq Christchurch
Aero,20100101,2259,9.9,24,D
Christchurch
Aero,20100102,2259,7.1,24,D
Christchurch
Aero,20100103,2259,1.8,24,D
Christchurch
Aero,20100104,2259,9.7,24,D
...
```

Empty line

Wanted data

cliflo.niwa.co.nz query result (csv)

Algorithm #1 for extracting data

- Many possibilities. One is:

```
skip lines until we get an empty line
skip two more lines
```

More robust against changes in file format than "skip 9 lines"

```
read a line
while line not empty:      # blank lines terminate actual data
    rows split line into pieces separated by comma
    date = piece[1]
    get month and day from date
    sunshine = float(piece[3])
    process month, day, sunshine data point (e.g., write to another file)
    read a line
```

Code

```
infile = open("junk.txt")   infile: <_io.TextIOWrapper name='junk.txt' mode='r' encoding='cp1252'>
line = infile.readline()   line: 'Christchurch Aero,20100103,2259,1.8,24,D\n'
while line != '\n':
    line = infile.readline()

infile.readline()
infile.readline()

line = infile.readline()
while line != '\n':
    pieces = line.split(',')   pieces: ['Christchurch Aero', '20100102', '2259', '7.1', '24', 'D\n']
    date = pieces[1]   date: '20100102'
    month = int(date[4:6])   month: 1
    day = int(date[6:8])   day: 2
    sunshine = float(pieces[3])   sunshine: 7.1
    print(month, day, sunshine)
    line = infile.readline()

infile.close()
```

```python
def process_data_line(line):
    pieces = line.split(',')
    date = pieces[1]
    month = int(date[4:6])
    day = int(date[6:8])
    sunshine = float(pieces[3])
    print(month, day, sunshine)

infile = open("junk.txt")
line = infile.readline()
while line != '\n':
    line = infile.readline()

infile.readline()
infile.readline()

line = infile.readline()
while line != '\n':
    process_data_line(line)
    line = infile.readline()

infile.close()
```

Variant using a function for data line processing

Question: what happens if the data file does not contain the expected two blank lines?

Algorithm #2 for extracting data

- Another is:

  get a list of all lines in file
  make a list of all those lines beginning "Christchurch Aero" for each of those lines:

      split line into pieces separated by comma
      date = piece[1]
      get month and day from date
      sunshine = float(piece[3])
      process month, day, sunshine data point (e.g., write to another file)

Simpler (?) but only works for this one base station.
Also, cannot handle huge files.

```python
def process_data_line(line):
    pieces = line.split(',')
    date = pieces[1]
    month = int(date[4:6])  # Extract month from date
    day = int(date[6:8])    # Extract day from date
    sunshine = float(pieces[3])  # Extract sunshine data
    print(month, day, sunshine)


infile = open("junk.txt", "r")  infile: <_io.TextIOWrapper
lines = infile.readlines()  lines: ['Name,Agent Number,Netw
infile.close()


for line in lines[6:]:  # Start from the 7th line (index 6)
    if line.startswith("Christchurch Aero"):
        process_data_line(line)
```

Algorithm #2b

- An improvement is to get the station name from line 7

```
get a list of all lines in file
station_name = start of line 7, up until ","
make a list of all lines beginning with station_name for each of
those lines:
    split line into pieces separated by comma
    date = piece[1]
    get month and day from date
    sunshine = float(piece[3])
    process month, day, sunshine data point (e.g., write to another file)
```

OK for any base station.
Still cannot handle huge files.

Writing output files (example)

```
# Open file for writing, prepare data
out_file = open('myoutput.txt', 'w')
data = "{0},{1},{2:.3f}\n".format(month, day, sunshine)

# Write data to file
# NB: must include newline character when using
write method
out_file.write(data)

# Close file
out_file.close()
```

writelines

```python
lines = ['First line', 'Second line', 'Third line']

with open('example.txt', 'w') as file:
    # method1?
    file.writelines(lines)
    # method2
    for line in lines:
        file.write(line)
```

First lineSecond lineThird line

```python
lines = ['First line\n', 'Second line\n', 'Third line\n']

with open('example.txt', 'w') as file:
    # method1
    file.writelines(lines)
    # method2
    for line in lines:
        file.write(line)
```

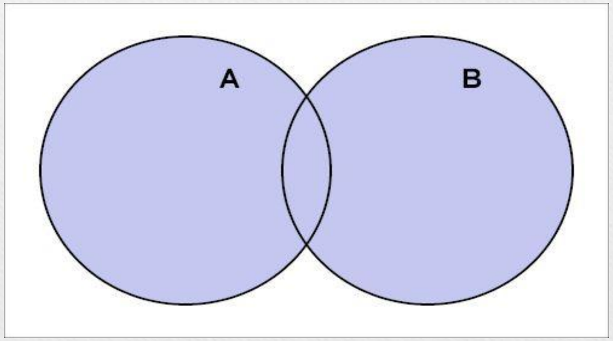
```
First line
Second line
Third line
```

Venn diagram

Union

- **union** creates a new set containing items in the object and/or in the argument

```
>>> print(household_pets.union(farmyard_animals))
{'cat', 'goat', 'dog', 'goldfish', 'gerbil', 'pig'}
```

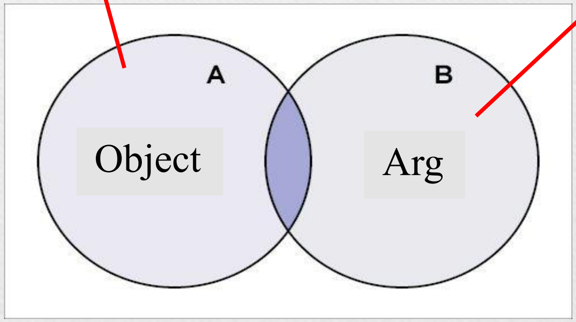



```
farmyard_animals = {"goat", "dog", "pig"}
household_pets = {"goldfish", "gerbil", "cat", "dog"}
```

Intersection

- **intersection** creates a new set containing items in both

```
>>> print(household_pets.intersection(farmyard_animals))
{'dog'}
```



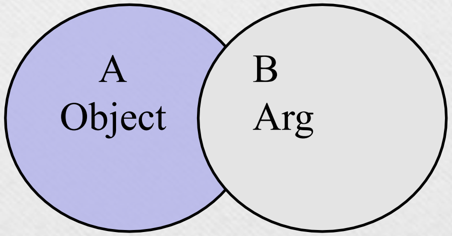

```
farmyard_animals = {"goat", "dog", "pig"}
household_pets = {"goldfish", "gerbil", "cat", "dog"}
```

Difference

```
farmyard_animals = {"goat", "dog", "pig"}
household_pets = {"goldfish", "gerbil", "cat", "dog"}
```

- **difference** creates a new set containing items in the object, but not in the argument

```
>>> print(household_pets.difference(farmyard_animals))
{'goldfish', 'gerbil', 'cat'}
>>> print(farmyard_animals.difference(household_pets))
{'goat', 'pig'}
```

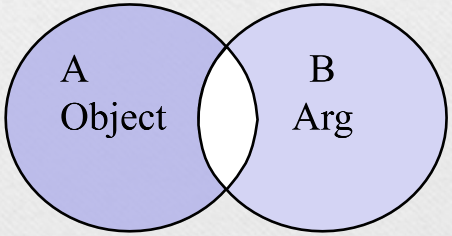


Symmetric difference

```
farmyard_animals = {"goat", "dog", "pig"}
household_pets = {"goldfish", "gerbil", "cat", "dog"}
```

- **`symmetric_difference`** creates a set with items in exactly one set (either but not both) (c.f. "exclusive or")

```
>>> print(household_pets.symmetric_difference(farmyard_animals))
{'pig', 'goldfish', 'gerbil', 'cat', 'goat'}
```



Modules, Namespaces,

- Writing modules
- Importing your own modules
- Documenting your modules
- Including test code in a module

Modules

- To control complexity, large programs are always broken into functions
  - Functional decomposition
- As programs get still larger, we need to further decompose the program into *modules*
  - One file per module
  - Each module contains a collection of functions (plus perhaps data)
- Any module can *import* the code and data from other modules
- The Python library is a large collection of modules

An example library module: math

- Can import the entire module

```
import math
print(math.pi)               # An imported data value
print(math.sqrt(23.456))     # An imported function
```

- Or we can import selected data/functions from module

```
from math import pi, sqrt
print(pi)
print(sqrt(23.456))
```

Using the circle module

- Just import it and use it!

```
import circle

r = 5.0
area = circle.area(r)
circum = circle.circumference(r)
...
```

- *import x* causes Python to **load and execute** the file *x.py*

Finding what's in a module

1. Read the *Python Standard Library* documentation
   – via *http://www.python.org/doc/*
2. In the shell window, import the module and type

   `help(moduleName)`, e.g., `help(math)`

   or, for its directory, `dir(math)`
3. Google, e.g., *python math module*
   – *https://docs.python.org/3/library/math.html*
   – For details on a particular function, can use the on-line help's index or type *help(moduleName.functionName)*

Create your modules

```
"""A module of functions related to circles."""

import math

def area(radius):
    """Returns the area of a circle given its
    radius."""
    return math.pi * radius**2
```

Output of `help(circle)` is now:

```
Help on module circle:

NAME
    circle - A module of functions related to circles.

FILE
    somewhere/MSE800/lectures/circle.py

FUNCTIONS
    area(radius)
        Return the area of a circle given its radius.
```

More on import

- When a module is *imported*, its `__name__` variable is set to the name of the module
- When a module is *run* as the main program, its `__name__` is set to "`__main__`"

```
if __name__ == '__main__':
```

- This line of code is very common in Python scripts and serves to determine whether a module is being run directly or imported into another module.

- __name__ is a built-in variable in Python, which is set to '__main__' when a module is run directly, and to the module name when it is imported.

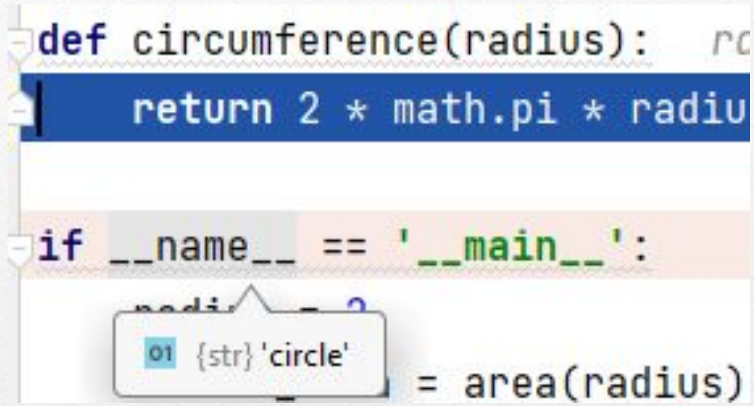
circle.py | main.py

```python
import math

def area(radius):
    return math.pi * radius ** 2

def circumference(radius):
    return 2 * math.pi * radius

if __name__ == '__main__':
    radius = 2
    circle_area = area(radius)
    circle_circumference = circumference(radius)
    print(f"Area: {circle_area}")
    print(f"Circumference: {circle_circumference}")
```



Using “__main__”:

circle.py | main.py

```python
import circle

def main():
    radius = 3
    circle_area = circle.area(radius)
    circle_circumference = circle.circumference(radius)
    print(f"Area: {circle_area}")
    print(f"Circumference: {circle_circumference}")

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

Pylint

- **Pylint** is a popular static code analysis tool for Python programming.
- It checks for errors in Python code, enforces a coding standard, and offers suggestions for refactoring and improving code quality.

Namespaces

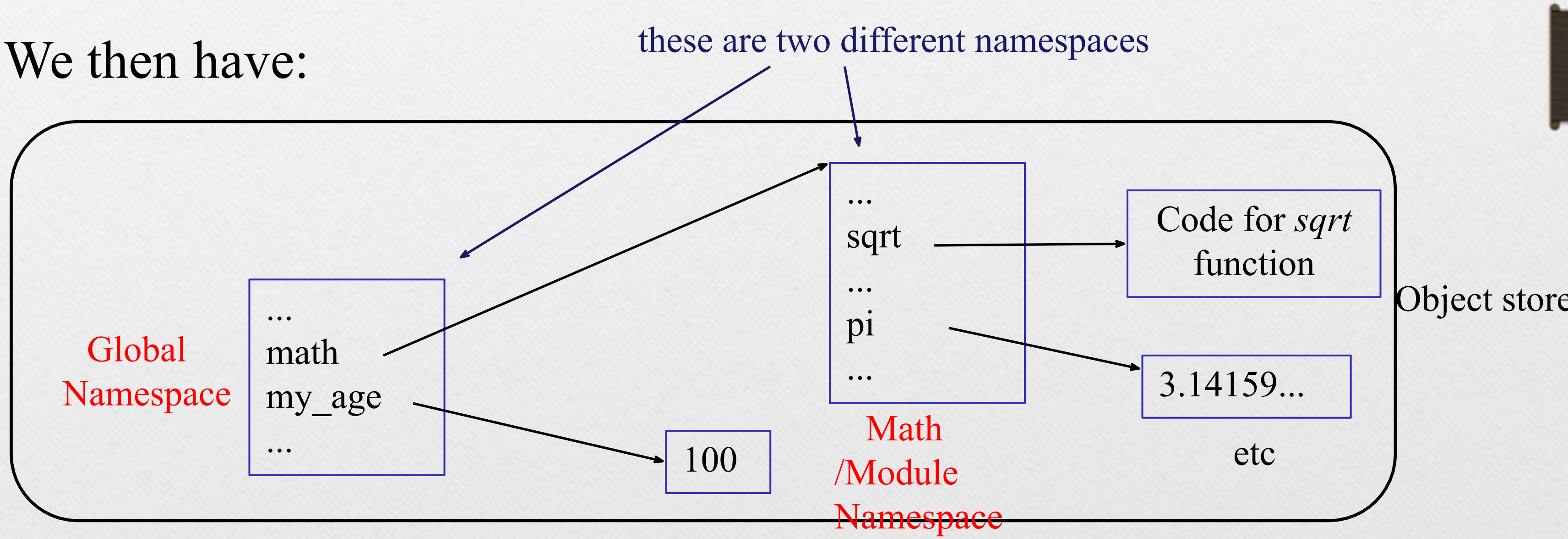
- Module namespace versus “local” namespace
- Function namespaces and the global namespace
- Global variables
- Global constants

Namespaces

- A namespace is just a “dictionary” of names
- Consider:

```
import math
my_age = 100
```

- We then have:



Local namespaces

A local namespace refers to the area where variables defined inside a function exist.

Local namespaces

- Consider the following nonsense program

```
import math

pi = 3  # A 'global' variable
print("Global pi =", pi)

def myfunc():
    pi = 4  # A local variable
    print("Pi in myfunc =", pi)


myfunc()

print("Global pi =", pi)
print("Math pi = ", math.pi)
```

Local namespaces

Output is:

```
Global pi = 3 Pi
in myfunc = 4
Global pi = 3
Math pi =
 3.141592653589793
```

Using globals

- A function can "see" variables in the global namespace, e.g.

```
x = 10
def blah():
    print("Within blah, x =", x)  # Prints 10!
```

- But new variables created by assignment inside a function are added to the *local* namespace.

```
x = 10
def blah():
 x = 20
    print("Within blah, x =", x)  # Prints 20!
```

- Python looks first in local namespace, then in global namespace if name not found.

Using globals (cont'd)

- It's illegal to reference a global variable from within a function and then create a local one of the same name. For example, the following gives a runtime error:

```
x = 10
def blah():
    print(x)
    x = 20
blah()
```

UnboundLocalError: local variable 'x' referenced before assignment

Assigning to a global

- BUT can assign to a global variable using a *global* statement:

```
x = 10
def blah():
    global x
    print("In blah, x =", x)
    x = 20

print("Initially, x =", x)
blah()
print("Post-blah, x =", x)
```

- Output is:

```
Initially, x = 10
In blah, x = 10
Post-blah, x = 20
```

Use of global variables in MSE800

- A very simple rule:

```
x = 10
def blah():
    print(x)
```

DON’T

i.e., don’t read or write global variables from within a function body

BUT: global constants are GOOD!

Global constants are variables whose values are intended to remain unchanged throughout the program. They are typically defined at the top of a file or module.

Feature	Global Variable	Global Constant
Mutability	Mutable (can be changed)	Immutable (should not be changed)
Purpose	To store data that can change	To define fixed values
Convention	Lowercase or camelCase naming	Uppercase naming

- **Style rule: avoid <u>magic numbers</u> in code**

```
for i in range(52):
…
theta = delta * 1.5707963267948966
…
while error > 0.000001:
```

- It's not obvious what all these numbers are.
- Instead give them ALL_CAPS names at the top of the module
- These are called *global constants*
  - Yes, pylint allows them :-)

global constants (cont’d)

- Instead write previous code as

```
WEEKS_IN_YEAR = 52
PI_OVER_TWO = math.pi / 2

ERROR_TOLERANCE = 0.000001
```

Global constants
(at top of module)

```
…

for i in range(WEEKS_IN_YEAR):

…

theta = delta * PI_OVER_TWO

…

while e > ERROR_TOLERANCE:
```

Ok, we already talked about functions in our previous classes, right?

Now! Let’s learn more about it!!!

More on functions

- Default parameters
- Named parameters
- Variable numbers of parameters

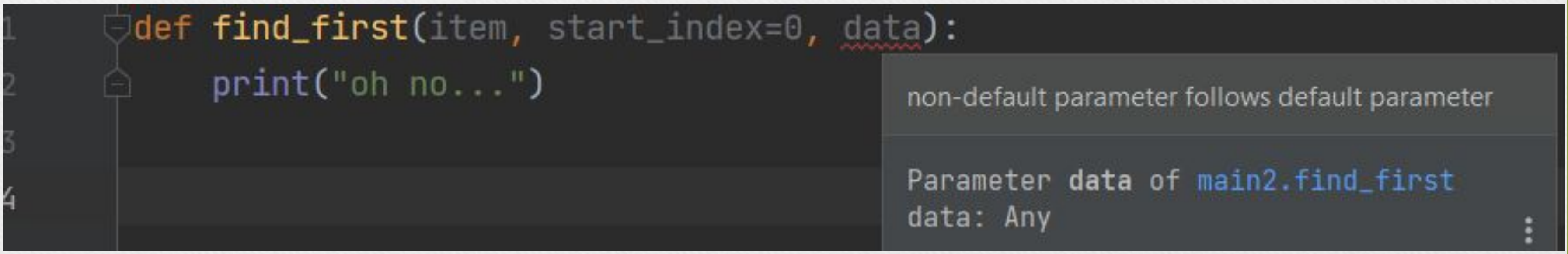
Default Parameters

```
deffind_first(item, data, start_index=0 ):
    """ The index of the first occurrence of item in data,
        starting at start_index or -1 if not found """
    index = -1
    for i in range(start_index, len(data)):
        if data[i] == item and index == -1:
            index = i
    return index
```

- Arguments are matched to parameters left to right
- Parameters without defaults must come first (see over)

Default Parameters (cont'd)
Left to right argument matching

```
def     find_first(item, start_index=0, data):# ILLEGAL
""" The index of the first occurrence of item in data, ... """
    ... etc ...
```



Default Parameters (cont'd)

Default values must be known at definition time

```
# LEGAL! Default value known at time of definition
def add_vals_or_six(x, y=2*3):
    return x + y

# ILLEGAL! Value of x unknown at time of definition
# (unless it's a global)
def add_vals_or_double_x(x, y=2*x):
    return x + y
```

Variable Parameter Lists

- How about a function that receives *any* number of arguments?
- For example, the Python function `max`?

```
>> max(1, 2, 3, -34, 453, 22)
453
```

- Achieved with a variable parameter list

```
def our_max(first, *rest
    ): biggest = first
    for value in rest:
        if value >
            biggest:
            biggest =
            value
    return biggest
```

```
>>> our_max(5, 4, 3, 2, 43,
2)
43
```

rest is a tuple containing all unmatched parameters. Can only have one such parameter

Keyword Arguments

- It's possible to assign arguments to parameters by naming them, instead of using their order - called "keyword arguments"

```
def describe_creature(name, species, age, weight):
  print('{} ({}): {} yrs, {} kg'.format(name, species, age, weight))
>>> # arguments by order
>>> describe_creature("Frodo", "Hobbit", 122, 40)
Frodo (Hobbit): 122 yrs, 40 kg
>>> # arguments by name
>>> describe_creature(age=122, weight=40, name="Frodo", species="Hobbit")
Frodo (Hobbit): 122 yrs, 40 kg
```

- Illegal to use a non-keyword argument after a keyword one

```
>>> describe_creature(age=122, weight=40, "Frodo", species="Hobbit")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
non-keyword arg after keyword arg: <string>, line 1
```

Quiz on default, variable, and keyword parameters

- Legal (and if so, what is the output), or illegal (if so, why)?

```
>>> def foo(x=42):
        print(x)
>>> foo(15)
>>> foo(15, 30)
>>> foo()
>>> foo(y=50)
>>> foo(x=10)

>>> def foo2(x, y=10, *z):
        print(x, y, z)
>>> foo2()
>>> foo2(20)
>>> foo2(20, 15)
>>> foo2(10, 20, 30, 40,
50)

>>> def foo3(y=10, x):
        print(y, x)
```

Quiz on default, variable, and named parameters

- Legal (and output), or illegal?

```
>>> def foo4(a, b, c, d, e=12):
      print(a, b, c, d, e)

>>> foo4(e=5, a=1, c=3, b=2, d=4)
>>> foo4(d=4, b=2, a=1, c=3)
>>> foo4(c=3, e=5, a=1, d=4)
```

**kwargs

[not officially in course, but nice to know (?)]

- We've seen **args* used to capture all remaining non-keyword arguments. *rest: a variable name, you can use any name you want
  - It's a *tuple*
- There's also ***kwargs* to capture all remaining keyword arguments
  - It's a *dictionary*

```
def kwarg_demo(**kwargs):
    print(kwargs)

kwarg_demo(name="Fred", age=20)
```

Prints

```
{'age': 20, 'name': 'Fred'}
```

Thank you