Python Seminar

- Needed Applications
  - Chrome (website: c9.io)

- GradQuant Resources
  - http://gradquant.ucr.edu/workshop-resources/

- Audience
  - Minimal programing experience.
  - Basic Python understanding.
    - or
  - Attended previous Python seminars.

This is part 3 of 3 Python seminars.

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Advanced Python

Part 3

Presented by GradQuant

Objectives

Part 1
- Variables (int, float)
- Math (+, -, *, /, %, **)
- Conditional Expressions

Part 2
- Strings (input, manipulation, and formatting)
- Lists
- Control Flow (Loops and Branches)

Part 3
- Sets and Dictionaries
- Functions
- Files

Sets

Ruth Anderson
CSE 140
University of Washington

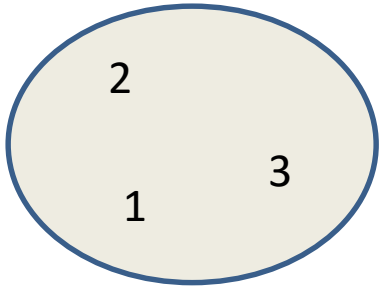
Sets

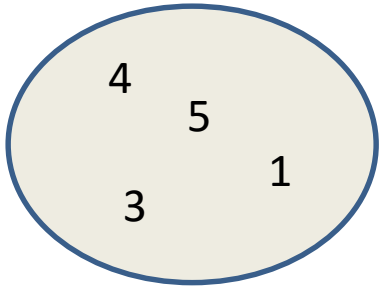
- Mathematical set: a collection of values, without duplicates or order
- Order does not matter

  { 1, 2, 3 } == { 3, 2, 1 }
- No duplicates

  { 3, 1, 4, 1, 5 } == { 5, 4, 3, 1 }
- For every data structure, ask:
  - How to create
  - How to query (look up) and perform other operations
    - (Can result in a new set, or in some other datatype)
  - How to modify

  Answer: http://docs.python.org/2/library/stdtypes.html#set





Two ways to create a set

1. Direct mathematical syntax:

```
odd = { 1, 3, 5 }
prime = { 2, 3, 5 }
```

Cannot express empty set: "**{ }**" means something else ☹

2. Construct from a **list:**

```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
empty = set([])
```

Python always **prints** using this syntax above

Set operations

```
odd = { 1, 3, 5 }
prime = { 2, 3, 5 }
```

- membership ∈ Python: **`in`** **`4 in prime`** ⇒ False
- union ∪ Python: **`|`** **`odd | prime`** ⇒ { 1, 2, 3, 5 }
- intersection ∩ Python: **`&`** **`odd & prime`** ⇒ { 3, 5 }
- difference \ or - Python: **`-`** **`odd - prime`** ⇒ { 1 }

Think in terms of **set operations**,
not in terms of iteration and element operations

  - Shorter, clearer, less error-prone, faster

Although we can do iteration over sets:

```
# iterates over items in arbitrary order
for item in myset:
    …
```

Modifying a set

- **Add** one element to a set:

```
myset.add(newelt)
myset = myset | { newelt }
```

- **Remove** one element from a set:

```
myset.remove(elt)    # elt must be in myset or raises err
myset.discard(elt)   # never errs
myset = myset - { elt }
```

What would this do?

```
myset = myset - elt
```

- Choose and remove some element from a set:

```
myset.pop()
```

Practice with sets

```
z = {5,6,7,8}
y = {1,2,3,"foo",1,5}
k = z & y
j = z | y
m = y - z
z.add(9)
```

List vs. set operations (1)

Find the common elements **in both** list1 and list2:

```
out1 = []
for i in list2:
    if i in list1:
        out1 .append(i)

# We will learn about list comprehensions later
out1 = [i for i in list2 if i in list1]
```

Find the common elements in both set1 and set2:

```
set1 & set2
```

Much shorter, clearer, easier to write!

List vs. set operations (2)

Find the elements in **either** list1 or list2 (**or both**) (without duplicates):

```
out2 = list(list1)          # make a copy
for i in list2:
   if i not in list1:       # don't append elements already in out2
      out2.append(i)
```

OR

```
out2 = list1+list2
for i in out1:              # out1 (from previous example), common
                            #  elements in both lists
   out2.remove(i)           # Remove common elements
```

Find the elements in either set1 or set2 (or both):

```
set1 | set2
```

List vs. set operations (3)

Find the elements in **either list but <u>not</u> in both**:

```
out3 = []
for i in list1+list2:
    if i not in list1 or i not in list2:
        out3.append(i)
```

Find the elements in either set but not in both:

```
set1 ^ set2
```

Not every value may be placed in a set

- Set elements must be immutable values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- Goal: only set operations change the set
  - after "**`myset.add(x)`**", **`x in myset`** ⇒ True
  - **`y in myset`** always evaluates to the same value

  Both conditions should hold until **`myset`** itself is changed
- Mutable elements can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
myset = { list1 }
```
⇐ Hypothetical; actually illegal in Python

`list1 in myset` ⇒ True

`list3 in myset` ⇒ True

`list2.append("c")` ⇐ not modifying **`myset`** "directly"

`list1 in myset` ⇒ ??? modifying **`myset`** "indirectly" would lead to different results

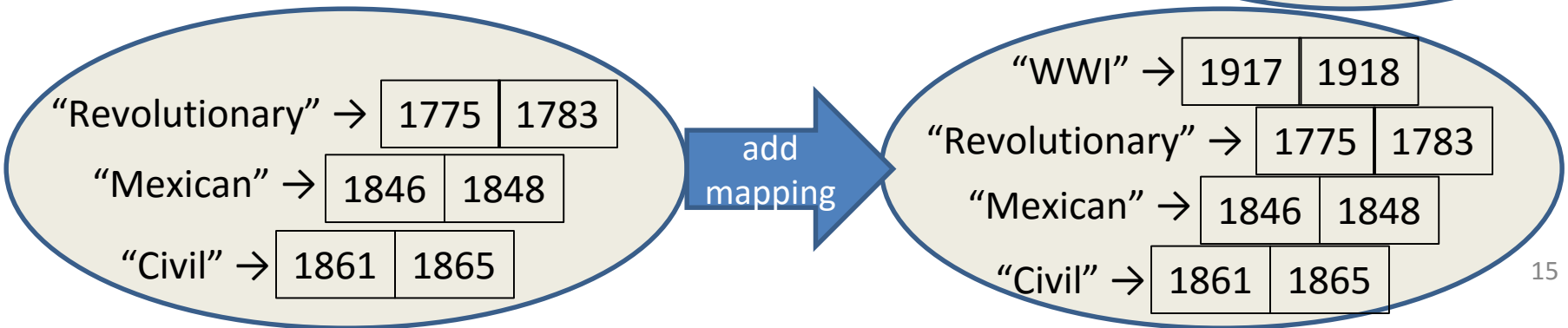
`list3 in myset` ⇒ ???

Dictionaries

Ruth Anderson
CSE 140
University of Washington

Dictionaries or mappings

- A dictionary maps each *key* to a *value*
- Order does not matter
- Given a key, can look up a value
  - Given a value, cannot look up its key
- **No duplicate keys**
  - Two or more keys may map to the same value
- *Keys* and *values* are Python values
  - Keys must be **immutable** (not a list, set, or dict)
- Can add *key* → *value* mappings to a dictionary
  - Can also remove (less common)



Dictionary syntax in Python

```
d = { }
d = dict()
```

Two different ways to create an empty dictionary

```
us_wars_by_end = {
  1783: "Revolutionary",
  1848: "Mexican",
  1865: "Civil" }
us_wars_by_name = {
  "Civil" : [1861, 1865],
  "Mexican" : [1846, 1848],
  "Revolutionary" : [1775, 1783]
}
```

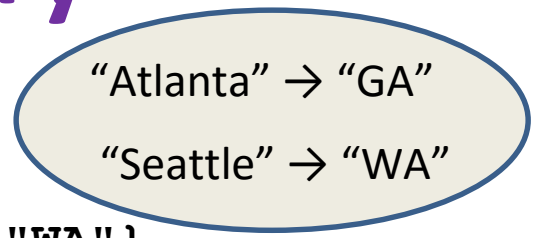
1783 → "Revolutionary"
1848 → "Mexican"
1865 → "Civil"

"Revolutionary" → 1775 | 1783
"Mexican" → 1846 | 1848
"Civil" → 1861 | 1865

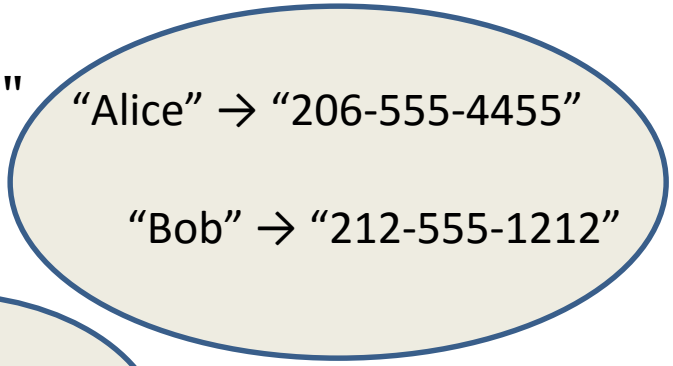
- Syntax just like arrays, for accessing and setting:

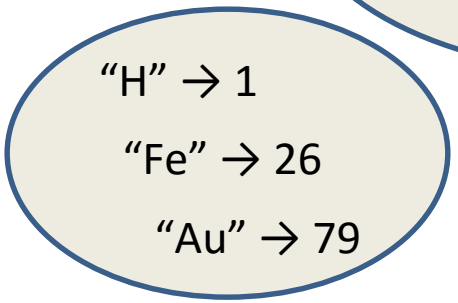
```
us_wars_by_end[1783]    ⇒
us_wars_by_end[1783][1:10]    ⇒
us_wars_by_name["WWI"] = [1917, 1918]
```

Creating a dictionary



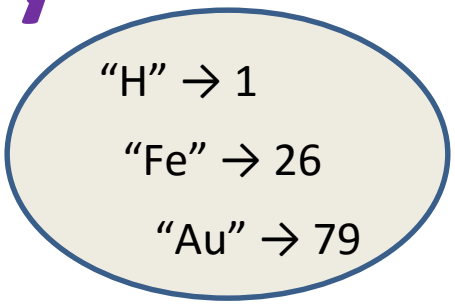

```
>>> state = {"Atlanta" : "GA", "Seattle" : "WA"}

>>> phonebook = dict()
>>> phonebook["Alice"] = "206-555-4455"
>>> phonebook["Bob"] = "212-555-2211"

>>> atomicnumber = {}
>>> atomicnumber["H"] = 1
>>> atomicnumber["Fe"] = 26
>>> atomicnumber["Au"] = 79
```





Accessing a dictionary

```
>>> atomicnumber = {"H":1, "Fe":26, "Au":79}
>>> atomicnumber["Au"]
79
>>> atomicnumber["B"]
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    atomicnumber["B"]
KeyError: 'B'
>>> atomicnumber.has_key("B")
False
>>> atomicnumber.keys()
['H', 'Au', 'Fe']
>>> atomicnumber.values()
[1, 79, 26]
>>> atomicnumber.items()
[('H', 1), ('Au', 79), ('Fe', 26)]
```



Good for iteration (for loops)

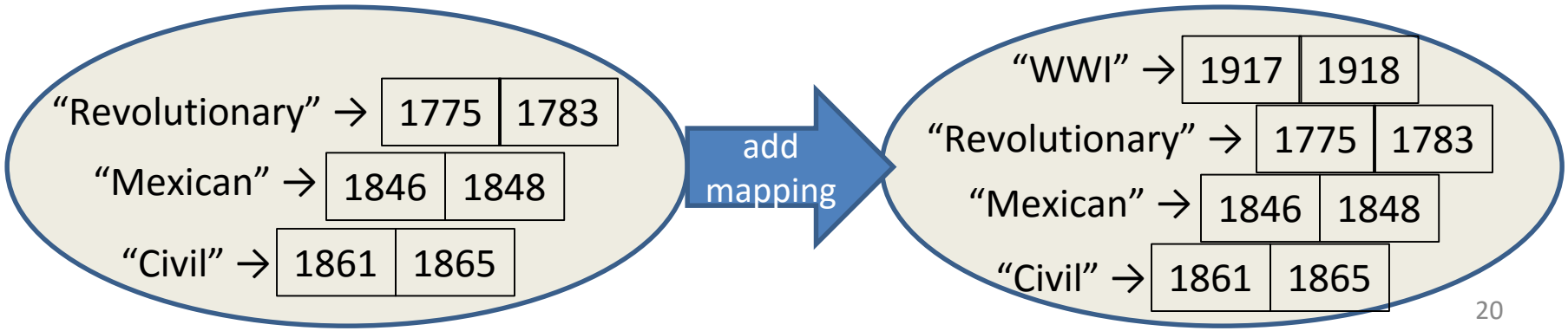
```
for key in mymap.keys():
  val = mymap[key]
  … use key and val

for key in mymap:
  val = mymap[key]
  … use key and val

for (key,val) in mymap.items():
  … use key and val
```

Iterating through a dictionary

```
atomicnumber = {"H":1, "Fe":26, "Au":79}

# Print out all the keys:
for element_name in atomicnumber.keys():
    print element_name
# Another way to print out all the keys:
for element_name in atomicnumber:
    print element_name

# Print out the keys and the values
for (element_name, element_number) in atomicnumber.items():
    print "name:",element_name, "number:",element_number
```

Modifying a dictionary

```
us_wars1 = {
    "Revolutionary" : [1775, 1783],
    "Mexican" : [1846, 1848],
    "Civil" : [1861, 1865] }

us_wars1["WWI"] = [1917, 1918]   # add mapping
del us_wars_by_name["Mexican"]   # remove mapping
```



Dictionary exercises

- Convert a list to a dictionary:
  - Given [5, 6, 7], produce {5:25, 6:36, 7:49}
- Reverse key with value in a dictionary:
  - Given {5:25, 6:36, 7:49}, produce {25:5, 36:6, 49:7}
- What does this do?

```
squares = { 1:1, 2:4, 3:9, 4:16 }
squares[3] + squares[3]
squares[3 + 3]
squares[2] + squares[2]
squares[2 + 2]
```

Dictionary exercise Solutions

- Convert a list to a dictionary:
  - E.g. Given [5, 6, 7], produce {5:25, 6:36, 7:49}

```
d = {}
for i in [5, 6, 7]:  # or range(5, 8)
    d[i] = i * i
```

- Reverse key with value in a dictionary:
  - E.g. Given {5:25, 6:36, 7:49}, produce {25:5, 36:6, 49:7}

```
k ={}
for i in d.keys():
    k[d[i]] = i
```

A list is like a dictionary

- A list maps an integer to a value
  - The integers must be a continuous range 0..*i*

```
mylist = ['a', 'b', 'c']
mylist[1] ⇒ 'b'
mylist[3] = 'c'    # error!
```

- In what ways is a list more convenient than a dictionary?
- In what ways is a list less convenient than a dictionary?

Not every value is allowed to be a key

- Keys must be immutable values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- Goal: only dictionary operations change the keyset
  - after "**`mydict[x] = y`**", **`mydict[x]`** ⇒ y
  - if **`a == b`**, then **`mydict[a] == mydict[b]`**

  These conditions should hold until **`mydict`** itself is changed
- Mutable keys can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
mydict = {}
mydict[list1] = "z"
mydict[list3] ⇒ "z"
list2.append("c")
mydict[list1] ⇒ ???
mydict[list3] ⇒ ???
```

⇐ Hypothetical; actually illegal in Python

Functions and abstraction

Ruth Anderson

UW CSE 140

Winter 2014

Functions

- In math, you use functions: sine, cosine, …
- In math, you define functions: $f(x) = x^2 + 2x + 1$

- A function packages up and names a computation
- Enables re-use of the computation (generalization)
- **D**on't **R**epeat **Y**ourself (DRY principle)
- Shorter, easier to understand, less error-prone

- Python lets you use and define functions
- We have already seen some Python functions:
  - **`len`**, **`float`**, **`int`**, **`str`**, **`range`**

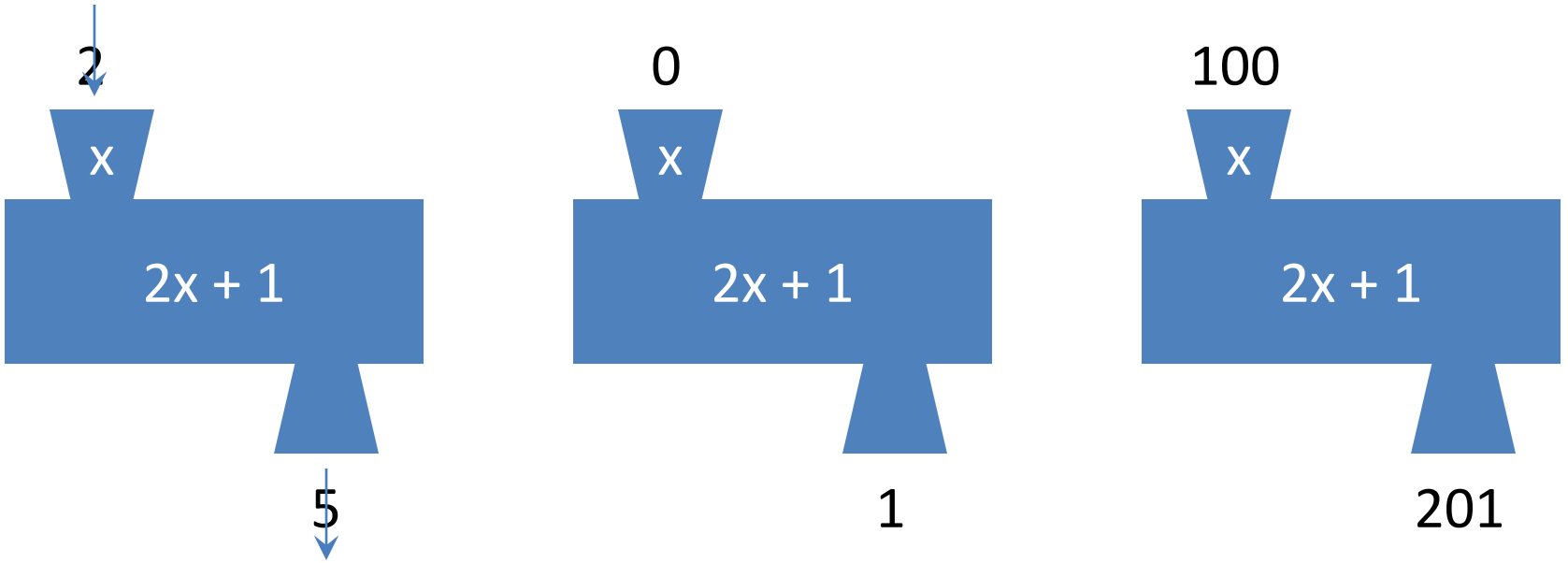
Using ("calling") a function

```
len("hello")                len("")
round(2.718)                round(3.14)
pow(2, 3)                       range(1, 5)
math.sin(0)
    math.sin(math.pi/2)
```

- Some need no input: **`random.random()`**
- All produce output
- What happens if you forget the parentheses on a function call? **`random.random`**
  - Functions are values too
  - Types we know about:
    int, float, str, bool, list, function

A function is a machine

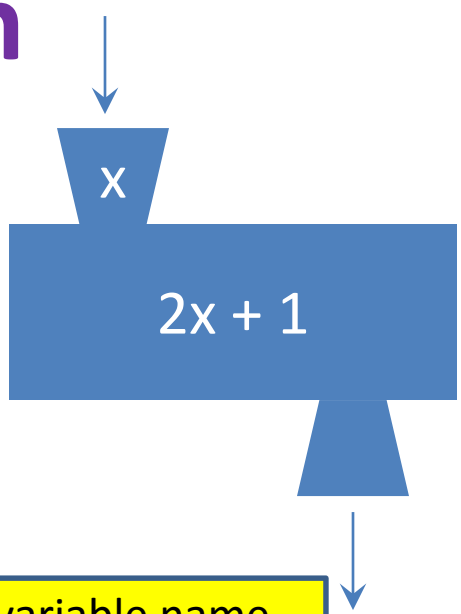
- You give it input
- It produces a result



In math: func(x) = 2x + 1

Creating a function

Define the machine,
including the input and the result



Name of the function.
Like "y = 5" for a variable

Keyword that means:
I am **def**ining a function

Input variable name,
or "formal parameter"

```
def dbl_plus(x):
    return 2*x + 1
```

Keyword that means:
This is the result

Return expression
(part of the **return** statement)

More function examples

Define the machine, including the input and the result

```
def square(x):
  return x * x

def fahr_to_cent(fahr):
  return (fahr - 32) / 9.0 * 5

def cent_to_fahr(cent):
  result = cent / 5.0 * 9 + 32
  return result

def abs(x):
  if x < 0:
    return - x
  else:
    return x
```

```
def print_hello():
  print "Hello, world"

def print_fahr_to_cent(fahr):
  result = fahr_to_cent(fahr)
  print result
```

No **return** statement
Returns the value
None
Executed for side effect

What is the result of:

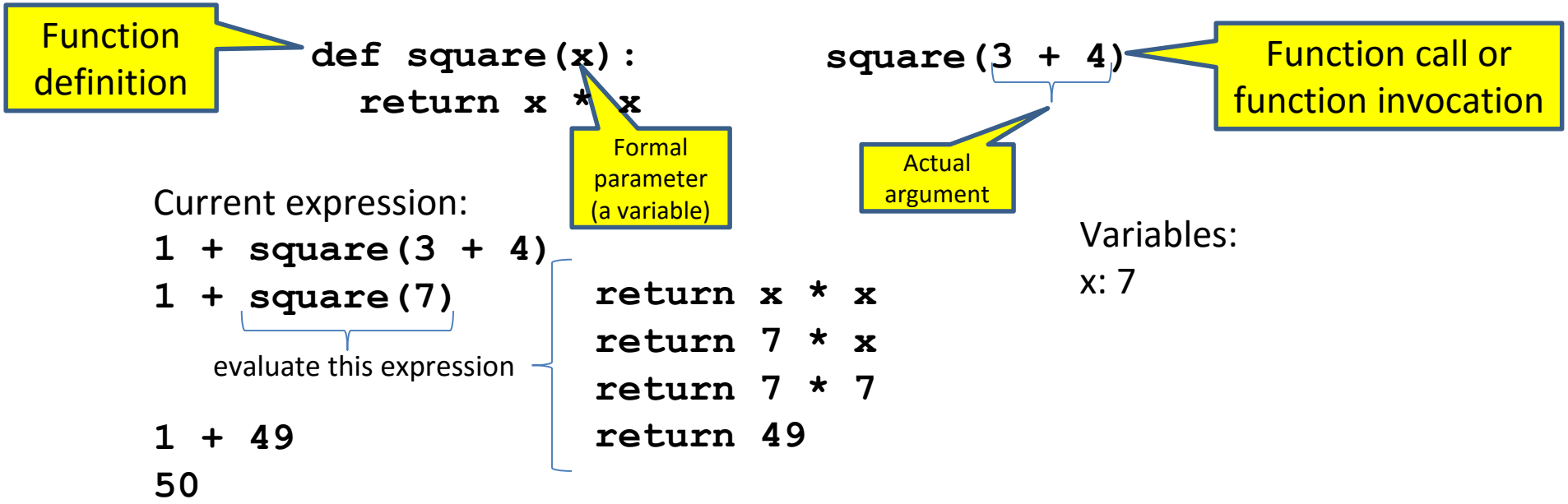
```
x = 42
square(3) + square(4)
print x
boiling = fahr_to_cent(212)
cold = cent_to_fahr(-40)
print result
print abs(-22)
print print_fahr_to_cent(32)
```

Digression: Two types of output

- An expression evaluates to a value
  - Which can be used by the containing expression or statement
- A **`print`** statement writes text to the screen

- The Python interpreter (command shell) reads statements and expressions, then executes them
- If the interpreter executes an expression, it prints its value
- In a program, evaluating an expression does not print it
- In a program, printing an expression does not permit it to be used elsewhere

How Python executes a function call



1. Evaluate the argument (at the call site)
2. Assign the formal parameter name to the argument's value
   - A *new* variable, not reuse of any existing variable of the same name
3. Evaluate the statements in the body one by one
4. At a return statement:
   - Remember the value of the expression
   - Formal parameter variable disappears – exists only during the call!
   - The call expression evaluates to the return value

Example of function invocation

```
def square(x):
  return x * x
```

```
square(3) + square(4)
return x * x
return 3 * x
return 3 * 3
return 9
9 + square(4)
      return x * x
      return 4 * x
      return 4 * 4
      return 16
9 + 16
25
```

Variables:

(none)
x: 3
x: 3
x: 3
x: 3
(none)
x: 4
x: 4
x: 4
x: 4
(none)
(none)

Expression with nested function invocations: Only one executes at a time

```
def fahr_to_cent(fahr):
  return (fahr - 32) / 9.0 * 5

def cent_to_fahr(cent):
  return cent / 5.0 * 9 + 32
```

```
fahr_to_cent(cent_to_fahr(20))
                return cent / 5.0 * 9 + 32
                return 20 / 5.0 * 9 + 32
                return 68
fahr_to_cent(68)
return (fahr - 32) / 9.0 * 5
return (68 - 32) / 9.0 * 5
return 20
20
```

Variables:

(none)

cent: 20

cent: 20

cent: 20

(none)

fahr: 68

fahr: 68

fahr: 68

(none)

Expression with nested function invocations: Only one executes at a time

```
def square(x):
  return x * x
```

	Variables:
`square(square(3))`	(none)
`return x * x`	x: 3
`return 3 * x`	x: 3
`return 3 * 3`	x: 3
`return 9`	x: 3
`square(9)`	(none)
`return x * x`	x: 9
`return 9 * x`	x: 9
`return 9 * 9`	x: 9
`return 81`	x: 9
`81`	(none)

Function that invokes another function: Both function invocations are active

```
def square(z):
  return z*z
def hypotenuse(x, y):
  return math.sqrt(square(x) + square(y))

hypotenuse(3, 4)
  return math.sqrt(square(x) + square(y))
  return math.sqrt(square(3) + square(y))
    return z*z
    return 3*3
    return 9
  return math.sqrt(9 + square(y))
  return math.sqrt(9 + square(4))
    return z*z
    return 4*4
    return 16
  return math.sqrt(9 + 16)
  return math.sqrt(25)
  return 5
5
```

Variables:

(none)

x: 3 y:4

x: 3 y:4

z: 3 x: 3 y:4

z: 3 x: 3 y:4

z: 3 x: 3 y:4

x: 3 y:4

x: 3 y:4

z: 4 x: 3 y:4

z: 4 x: 3 y:4

z: 4 x: 3 y:4

x: 3 y:4

x: 3 y:4

x: 3 y:4

(none)

Shadowing of formal variable names

```
def square(x):
  return x*x
def hypotenuse(x, y):
  return math.sqrt(square(x) + square(y))

hypotenuse(3, 4)
  return math.sqrt(square(x) + square(y))
  return math.sqrt(square(3) + square(y))
    return x*x
    return 3*3
    return 9
  return math.sqrt(9 + square(y))
  return math.sqrt(9 + square(4))
    return x*x
    return 4*4
    return 16
  return math.sqrt(9 + 16)
  return math.sqrt(25)
  return 5
5
```

Same formal parameter name

Variables:

(none)

x: 3 y:4

x: 3 y:4

x: 3 x: 3 y:4

x: 3 x: 3 y:4

x: 3 x: 3 y:4

x: 3 y:4

x: 3 y:4

x: 4 x: 3 y:4

x: 4 x: 3 y:4

x: 4 x: 3 y:4

x: 3 y:4

x: 3 y:4

x: 3 y:4

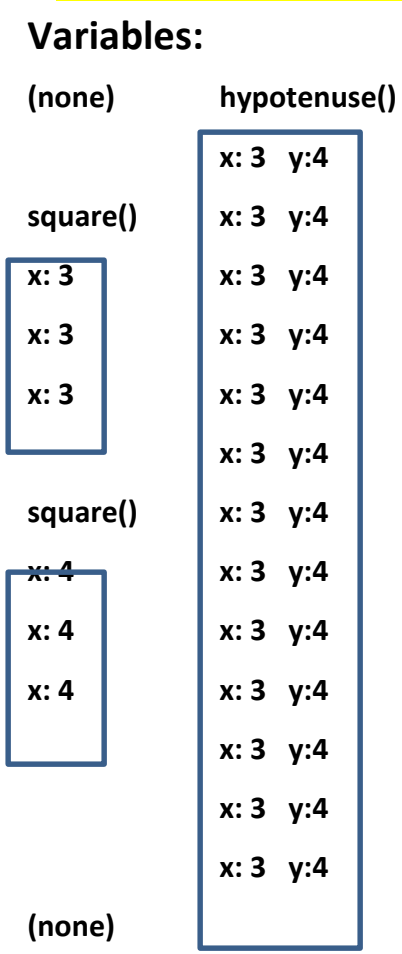
(none)

Formal parameter is a *new* variable

Shadowing of formal variable names

```
def square(x):
  return x*x
def hypotenuse(x, y):
  return math.sqrt(square(x) + square(y))

hypotenuse(3, 4)
  return math.sqrt(square(x) + square(y))
  return math.sqrt(square(3) + square(y))
    return x*x
    return 3*3
    return 9
  return math.sqrt(9 + square(y))
  return math.sqrt(9 + square(4))
    return x*x
    return 4*4
    return 16
  return math.sqrt(9 + 16)
  return math.sqrt(25)
  return 5
5
```

Same diagram, with *variable scopes* or *environment frames* shown explicitly



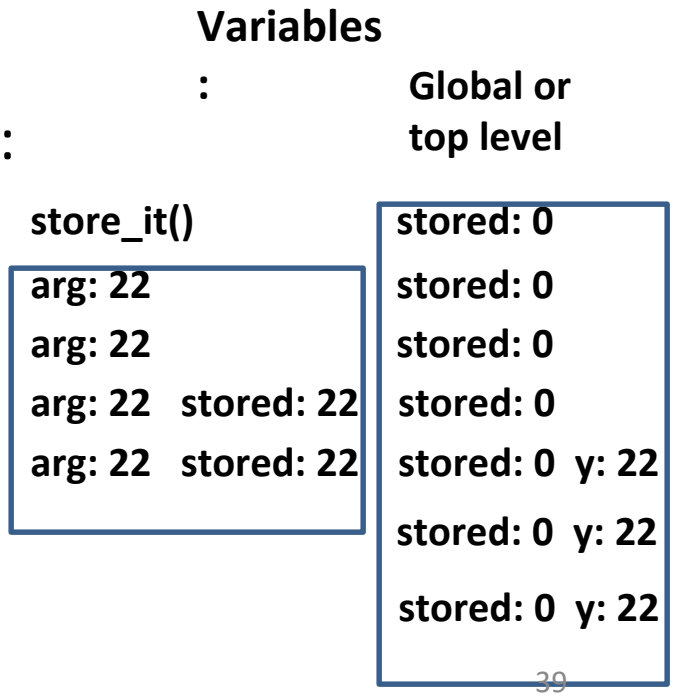
In a function body, assignment creates a temporary variable (like the formal parameter)

```
stored = 0
def store_it(arg):
  stored = arg
  return stored
y = store_it(22)
print y
print stored
```

★ ★

Show evaluation of the starred expressions:

```
y = store_it(22)
      stored = arg; return stored
      stored = 22; return stored
      return stored
      return 22
y = 22
print stored
print 0
```



How to look up a variable

Idea: find the nearest variable of the given name

1. Check whether the variable is defined in the local scope
2. … check any intermediate scopes (**none** in CSE 140!) …
3. Check whether the variable is defined in the global scope

If a local and a global variable have the same name, the global variable is inaccessible ("shadowed")

This is confusing; try to avoid such shadowing

```
x = 22
stored = 100
def lookup():
    x = 42
    return stored + x
lookup()
x = 5
stored = 200
lookup()
```

```
def lookup():
    x = 42
    return stored + x
x = 22
stored = 100
lookup()
x = 5
stored = 200
lookup()
```

What happens if we define **stored** *after* **lookup**?

Local variables exist only while the function is executing

```
def cent_to_fahr(cent):
    result = cent / 5.0 * 9 + 32
    return result

tempf = cent_to_fahr(15)
print result
```

Use only the local and the global scope

```
myvar = 1

def outer():
    myvar = 1000
    return inner()

def inner():
    return myvar

print outer()
```

The handouts have a more precise rule,
which applies when you define a function inside another function.

Abstraction

- Abstraction = ignore some details
- Generalization = become usable in more contexts
- Abstraction over computations:
  - functional abstraction, a.k.a. procedural abstraction
- As long as you know what the function means, you don't care how it computes that value
  - You don't care about the *implementation* (the function body)

Defining absolute value

```
def abs(x):
  if val < 0:
    return -1 * val
  else:
    return 1 * val
```

```
def abs(x):
  if val < 0:
    result = - val
  else:
    result = val
  return result
```

```
def abs(x):
  if val < 0:
    return - val
  else:
    return val
```

```
def abs(x):
  return math.sqrt(x*x)
```

Defining round (for positive numbers)

```
def round(x):
  return int(x+0.5)
```

```
def round(x):
  fraction = x - int(x)
  if fraction >= .5:
    return int(x) + 1
  else:
    return int(x)
```

Two types of documentation

1. Documentation for users/clients/callers
   - Document the *purpose* or *meaning* or *abstraction* that the function represents
   - Tells what the function does
   - Should be written for *every* function
2. Documentation for programmers who are reading the code
   - Document the *implementation* – specific code choices
   - Tells how the function does it
   - Only necessary for tricky or interesting bits of the code

For users: a string as the first element of the function body

For programmers: arbitrary text after #

```
def square(x):
    """Returns the square of its
argument."""
    # "x*x" can be more precise than "x**2"
    return x*x
```

Multi-line strings

- New way to write a string – surrounded by three quotes instead of just one
  - `"hello"`
  - `'hello'`
  - `"""hello"""`
  - `'''hello'''`
- Any of these works for a documentation string
- Triple-quote version:
  - can include newlines (carriage returns), so the string can span multiple lines
  - can include quotation marks

Don’t write useless comments

- Comments should give information that is not apparent from the code
- Here is a counter-productive comment that merely clutters the code, which makes the code *harder* to read:

```
# increment the value of x
x = x + 1
```

Where to write comments

- By convention, write a comment *above* the code that it describes (or, more rarely, on the same line)
  - First, a reader sees the English intuition or explanation, then the possibly-confusing code

```
# The following code is adapted from
# "Introduction to Algorithms", by Cormen et al.,
# section 14.22.
while (n > i):
    ...
```

- A comment may appear anywhere in your program, including at the end of a line:

```
x = y + x    # a comment about this line
```

- For a line that starts with **#**, indentation must be consistent with surrounding code

Each variable should represent one thing

```
def atm_to_mbar(pressure):
    return pressure * 1013.25

def mbar_to_mmHg(pressure):
    return pressure * 0.75006

# Confusing
pressure = 1.2 # in atmospheres
pressure = atm_to_mbar(pressure)
pressure = mbar_to_mmHg(pressure)
print pressure

# Better
in_atm = 1.2
in_mbar = atm_to_mbar(in_atm)
in_mmHg = mbar_to_mmHg(in_mbar)
print in_mmHg
```

```
# Best
def atm_to_mmHg(pressure):
    in_mbar = atm_to_mbar(pressure)
    in_mmHg = mbar_to_mmHg(in_mbar)
    return in_mmHg
print atm_to_mmHg(1.2)
```

Corollary: Each variable should contain values of only one type

```
# Legal, but confusing: don't do this!
x = 3
…
x = "hello"
…
x = [3, 1, 4, 1, 5]
…
```

If you use a descriptive variable name, you are unlikely to make these mistakes

Exercises

```
def cent_to_fahr(c):
  print cent / 5.0 * 9 + 32

print cent_to_fahr(20)
```

```
def myfunc(n):
  total = 0
  for i in range(n):
    total = total + i
  return total

print myfunc(4)
```

```
def c_to_f(c):
    print "c_to_f"
    return c / 5.0 * 9 + 32

def make_message(temp):
    print "make_message"
    return ("The temperature is "
+ str(temp))

for tempc in [-40,0,37]:
    tempf = c_to_f(tempc)
    message = make_message(tempf)
    print message
```

```
double(7)
```

```
abs(-20 - 2) + 20
```

Use the Python Tutor:
http://pythontutor.com/

What does this print?

```
def cent_to_fahr(cent):
    print cent / 5.0 * 9 + 32

print cent_to_fahr(20)
```

What does this print?

```
def myfunc(n):
    total = 0
    for i in range(n):
        total = total + i
    return total

print myfunc(4)
```

What does this print?

```
def c_to_f(c):
    print "c_to_f"
    return c / 5.0 * 9 + 32

def make_message(temp):
    print "make_message"
    return "The temperature is " + str(temp)

for tempc in [-40,0,37]:
    tempf = c_to_f(tempc)
    message = make_message(tempf)
    print message
```

```
c_to_f
make_message
The temperature is -40.0
c_to_f
make_message
The temperature is 32.0
c_to_f
make_message
The temperature is 98.6
```

Decomposing a problem

- Breaking down a program into functions is *the fundamental activity* of programming!
- How do you decide when to use a function?
  - One rule: DRY (Don't Repeat Yourself)
  - Whenever you are tempted to copy and paste code, don't!
- Now, how do you design a function?

How to design a function

1. Wishful thinking: Write the program as if the function already exists

2. Write a specification: Describe the inputs and output, including their types

No implementation yet!

3. Write tests: Example inputs and outputs

4. Write the function body (the implementation)

First, write your plan in English, then translate to Python

```
print "Temperature in Farenheit:", tempf
tempc = fahr_to_cent(tempf)
print "Temperature in Celsius:", tempc

def fahr_to_cent(f):
    """Input: a number representing degrees
Farenheit
    Return value: a number representing degrees
centigrade
    """
    result = (f - 32) / 9.0 * 5
    return result

assert fahr_to_cent(32) == 0
assert fahr_to_cent(212) == 100
assert fahr_to_cent(98.6) == 37
assert fahr_to_cent(-40) == -40
```

Review: how to evaluate a function call

1. Evaluate the function and its arguments to values
   - If the function value is not a function, execution terminates with an error
2. Create a new stack frame
   - The parent frame is the one where the function is defined
     - In CSE 140, this is always the global frame
   - A frame has bindings from variables to values
   - Looking up a variable starts here
     - Proceeds to the next older frame if no match here
     - The oldest frame is the "global" frame
     - All the frames together are called the "environment"
   - Assignments happen here
3. Assign the actual argument values to the formal parameter variable
   - In the new stack frame
4. Evaluate the body
   - At a return statement, remember the value and exit
   - If at end of the body, return **`None`**
5. Remove the stack frame
6. The call evaluates to the returned value

Functions are values
The function can be an expression

```
def double(x):
    return 2*x
print double
myfns = [math.sqrt, int, double, math.cos]
myfns[1](3.14)
myfns[2](3.14)
myfns[3](3.14)

def doubler():
    return double
doubler()(2.718)
```

File I/O

Ruth Anderson
UW CSE 140
Winter 2014

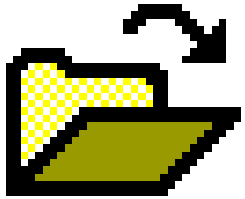
File Input and Output

- As a programmer, when would one use a file?
- As a programmer, what does one do with a file?

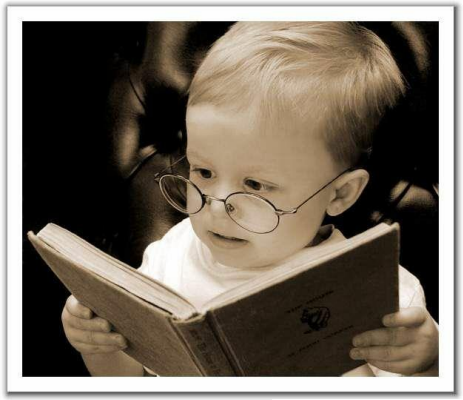
Files store information when a program is not running

Important operations:

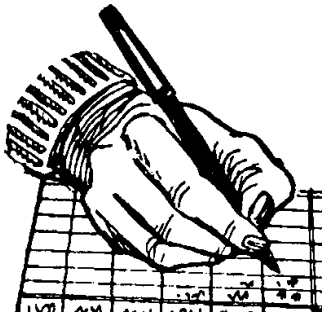
- open a file



- close a file



- read data
- write data



Files and filenames

- A file object represents data on your disk drive
  - Can read from it and write to it
- A filename (usually a string) states where to find the data on your disk drive
  - Can be used to find/create a file
  - Examples:
    - Linux/Mac:`"/home/rea/class/140/lectures/file_io.pptx"`
    - Windows:`"C:\Users\rea\My Documents\cute_dog.jpg"`
    - Linux/Mac: `"homework3/images/Husky.png"`
    - `"Husky.png"`

Two types of filenames

- An Absolute filename gives a specific location on disk:
  `"/home/rea/class/140/14wi/lectures/file_io.pptx"` or
  `"C:\Users\rea\My Documents\homework3\images\Husky.png"`
  - Starts with "/" (Unix) or "C:\" (Windows)
  - Warning: code will fail to find the file if you move/rename files or run your program on a different computer
- A Relative filename gives a location relative to the *current working directory*:
  `"lectures/file_io.pptx"` or `" images\Husky.png"`
  - Warning: code will fail to find the file unless you run your program from a directory that contains the given contents
- *A relative filename is usually a better choice*

Examples

Linux/Mac: These *could* all refer to the same file:

```
"/home/rea/class/140/homework3/images/Husky.png"
"homework3/images/Husky.png"
"images/Husky.png"
"Husky.png"
```

Windows: These *could* all refer to the same file:

```
"C:\Users\rea\My Documents\class\140\homework3\images\Husky.png"
"homework3\images\Husky.png"
"images\Husky.png"
"Husky.png"
```

“Current Working Directory” in Python

The directory from which you ran Python

To determine it from a Python program:

```
>>> import os   # "os" stands for "operating system"
>>> os.getcwd()
'/Users/johndoe/Documents'
```

Can be the source of confusion: where are my files?

Reading a file in python

```
# Open takes a filename and returns a file.
# This fails if the file cannot be found & opened.
myfile = open("datafile.dat")

# Approach 1:
for line_of_text in myfile:
   … process line_of_text

# Approach 2:
all_data_as_a_big_string = myfile.read()

myfile.close() # close the file when done reading
```

Assumption: file is a sequence of lines
Where does Python expect to find this file (note the relative pathname)?

Reading a file Example

```
# Count the number of words in a text file
in_file = "thesis.txt"
myfile = open(in_file)
num_words = 0
for line_of_text in myfile:
    word_list = line_of_text.split()
    num_words += len(word_list)
myfile.close()

print "Total words in file: ", num_words
```

Reading a file multiple times

You can iterate over a **list** as many times as you like:

```
mylist = [ 3, 1, 4, 1, 5, 9 ]
for elt in mylist:
  … process elt
for elt in mylist:
  … process elt
```

Iterating over a **file** uses it up:

```
myfile = open("datafile.dat")
for line_of_text in myfile:
   … process line_of_text
for line_of_text in myfile:
   … process line_of_text
```

This loop body will never be executed!

How to read a file multiple times?

Solution 1: Read into a list, then iterate over it

```
myfile = open("datafile.dat")
mylines = []
for line_of_text in myfile:
   mylines.append(line_of_text)
… use mylines
```

Solution 2: Re-create the file object (slower, but a better choice if the file does not fit in memory)

```
myfile = open("datafile.dat")
for line_of_text in myfile:
   … process line_of_text
myfile = open("datafile.dat")
for line_of_text in myfile:
   … process line_of_text
```

Writing to a file in python

```
# Replaces any existing file of this name
myfile = open("output.dat", "w")

# Just like printing output
myfile.write("a bunch of data")
myfile.write("a line of text\n")

myfile.write(4)
myfile.write(str(4))

myfile.close()
```

open for **W**riting (no argument, or "**r**", for **R**eading)

"\n" means end of line (**N**ewline)

Wrong; results in:
`TypeError: expected a character buffer object`

Right. Argument must be a string

close when done with all writing

Python Editors

- Eclipse with PyDev
  - http://pydev.org/
- Sublime Text
  - http://www.sublimetext.com/
- PyCharm
  - http://www.jetbrains.com/pycharm/

- Features
  - Free versions
  - Multiplatform
  - Python integration

Version Control

- Git
  - http://git-scm.com/
  - Common Public Repository
    - https://github.com/
    - https://bitbucket.org/
  - Software
    - http://www.sourcetreeapp.com/
    - Supported in many editors.
- Subversion (SVN)
  - http://subversion.apache.org/

Resources

- Python's website
  - http://www.python.org/
- Python Tutorial - Codecademy
  - http://www.codecademy.com/tracks/python
- GradQuant Resources
  - http://gradquant.ucr.edu/workshop-resources/
  - http://bit.ly/1KIJcEU (slides)
  - http://bit.ly/1Ew4FzZ (code examples)

- Google
  - Search for "python …"
- Stack Overflow website
  - http://stackoverflow.com/

GradQuant

- One-on-one Consultations
  - Make appointment on the website
  - http://gradquant.ucr.edu
- Python Seminars
  - Python Fundamentals (Part 1)
  - Data Manipulation with Python (Part 2)
  - Advanced Python (Part 3)