SUBVERSION , FUNCTIONS, PARAMETERS, AND FILE HANDLING

CSSE 120 – Rose-Hulman Institute of Technology

Outline

- Tools: Version Control
- Functions :
  - Math, Maple, Python
  - Function definition and invocation mechanics
  - Exercise: writing and invoking a function **`sumPowers`**
  - Nested function calls and execution order
  - Code-reading exercise
- Files
  - Opening, reading/writing, closing
- Begin *RobotPathViaPoints* exercise

Software Engineering Tools

- The computer is a powerful tool
- We can use it to make software development easier and less error prone!
- Some software engineering tools:
  - IDEs, like Eclipse and IDLE
  - Version Control Systems, like Subversion
  - Testing frameworks, like JUnit
  - Diagramming applications, like UMLet, Violet and Visio
  - Modeling languages, like Alloy, Z, and JML

Version Control Systems

- Store "snapshots" of all the changes to a project over time
- Benefits:
  - Multiple users
    - Multiple users can share work on a project
    - Record who made what changes to a project
    - Provide help in resolving conflicts between what the multiple users do
    - Maintain multiple different versions of a project simultaneously
  - Logging and Backups
    - Act as a "global undo" to whatever version you want to go back to
    - Maintain a log of the changes made
    - Can simplify debugging
  - Drop boxes are history!
    - Turn in programming projects
    - Get it back with comments from the grader embedded in the code

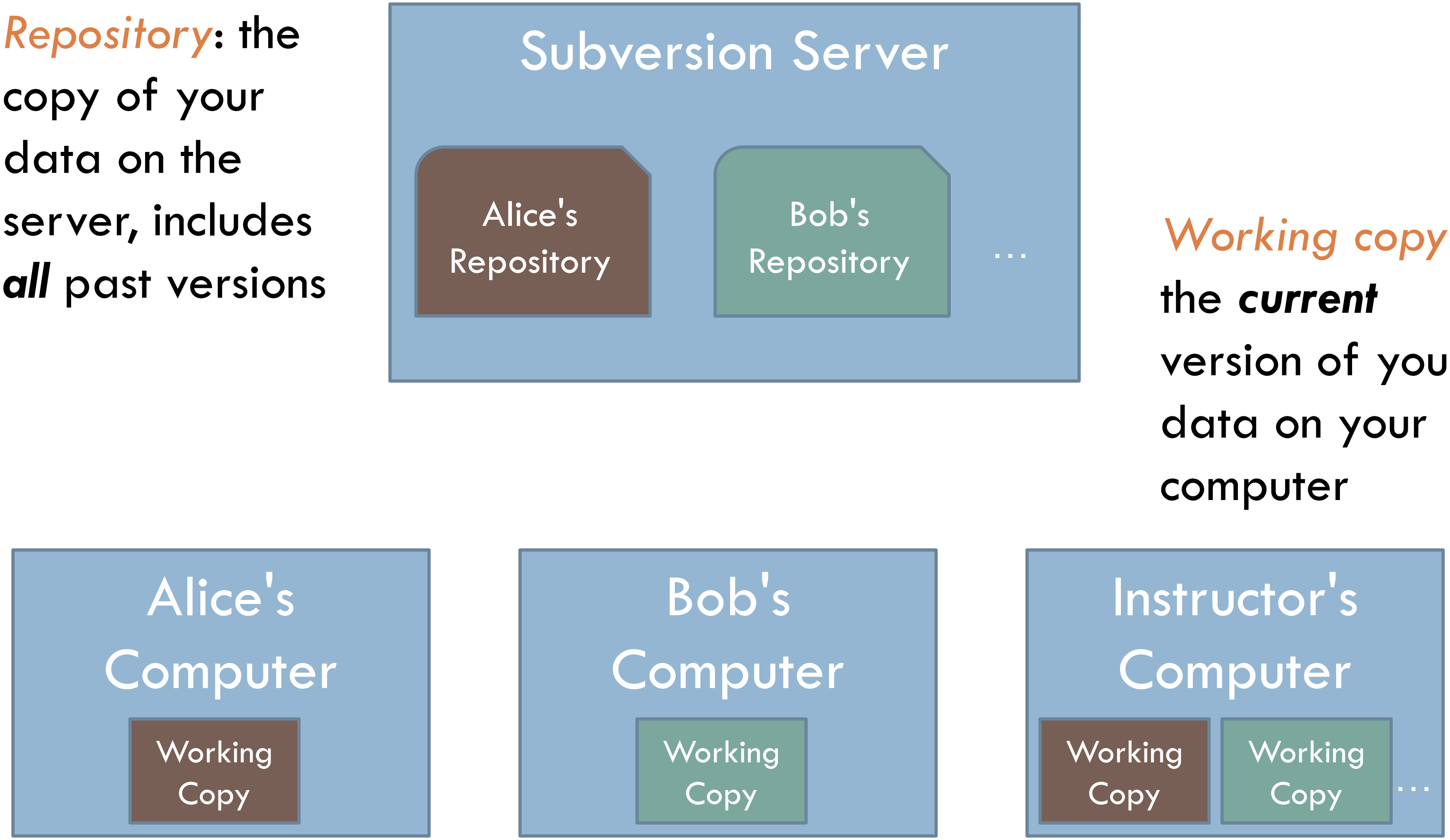
Our Version Control System

- Subversion, sometimes called SVN
- A free, open-source application
- Lots of tool support available
  - Works on all major computing platforms
  - **TortoiseSVN** for version control in Windows Explorer
  - **Subclipse** for version control inside Eclipse

Q1a

Version Control Terms

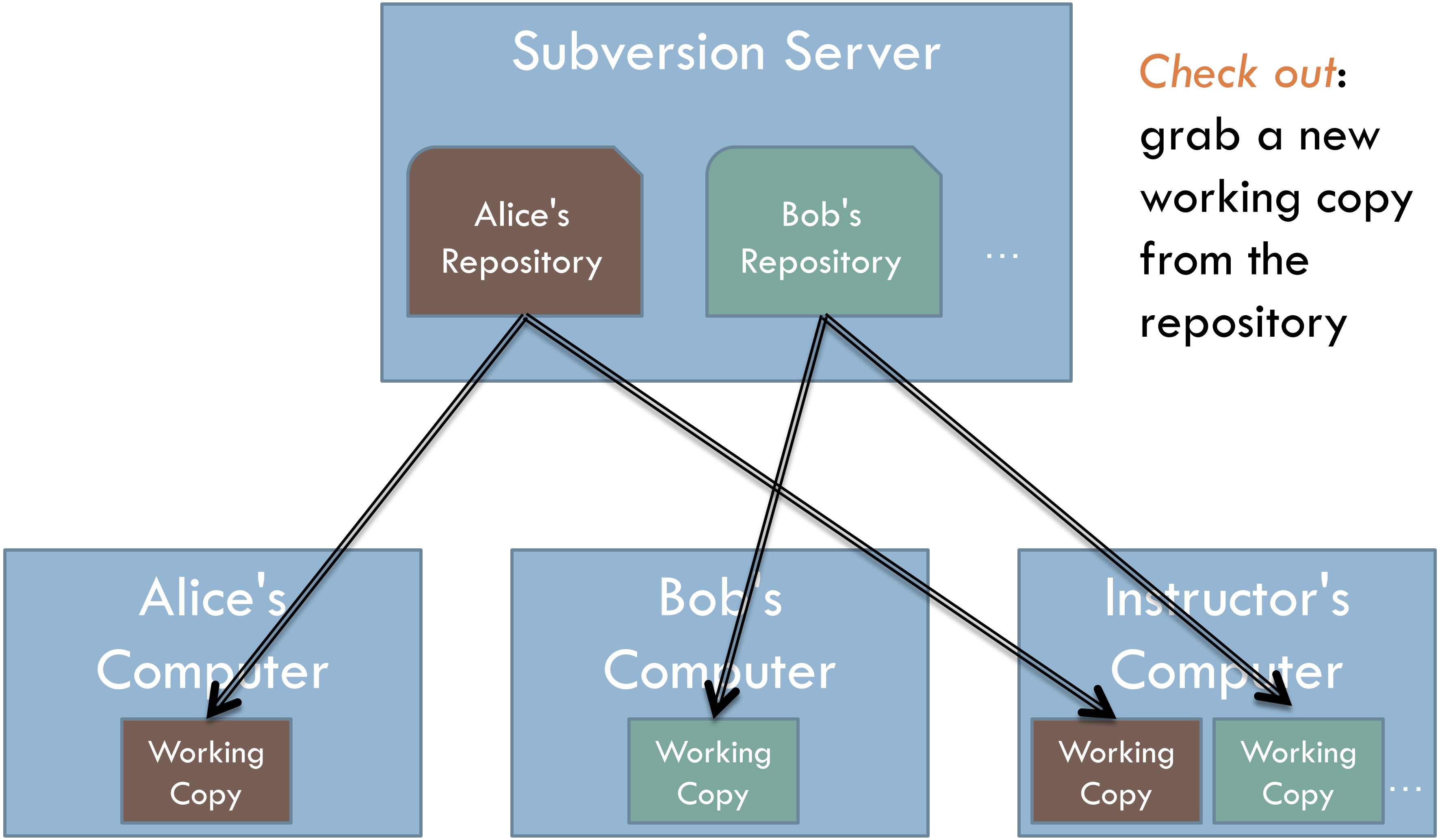
Repository: the copy of your data on the server, includes ***all*** past versions



Working copy: the ***current*** version of your data on your computer

Q1b

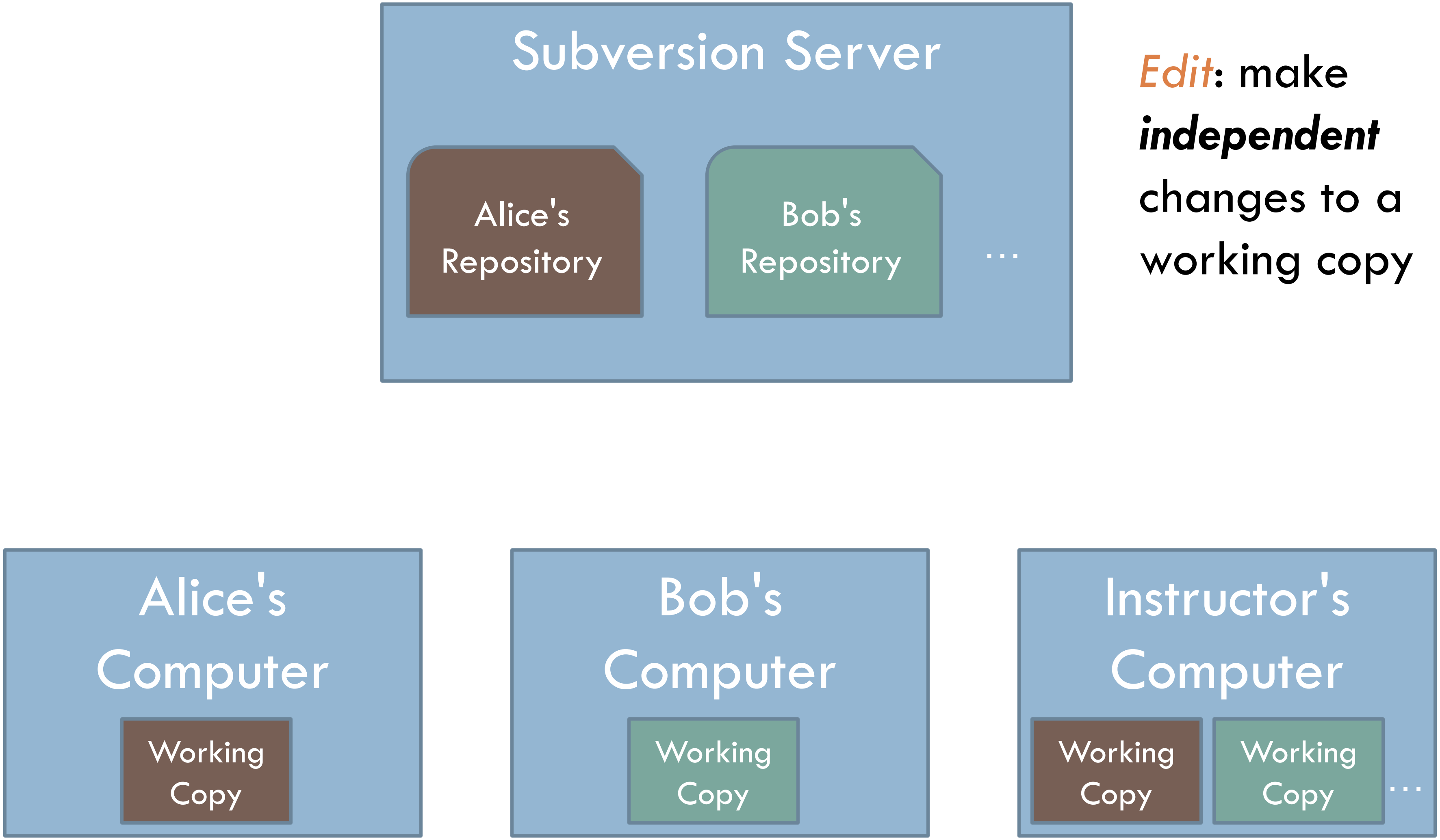
Version Control Steps—Check Out



Check out: grab a new working copy from the repository

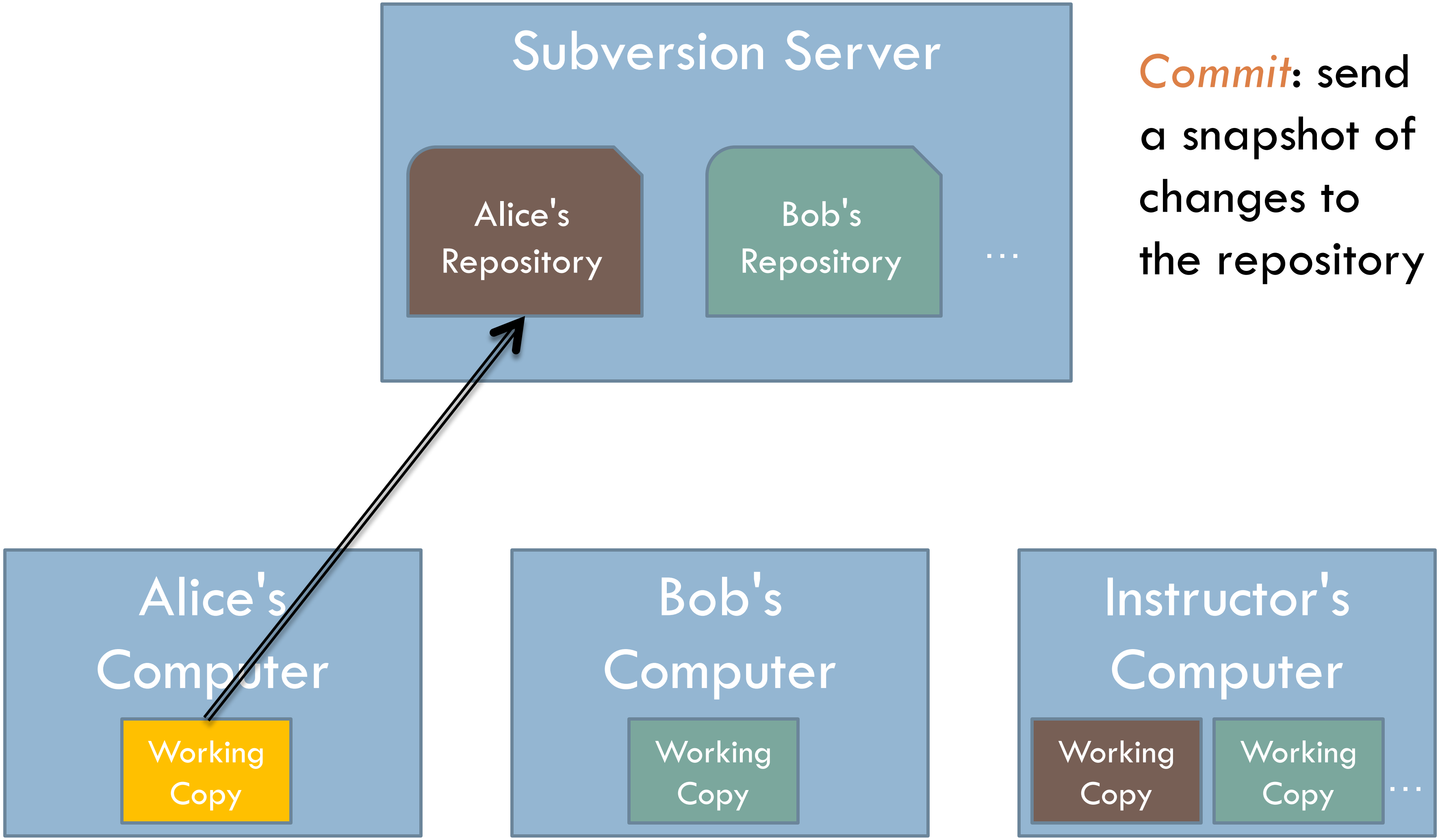
Q2a

Version Control Steps—Edit



Edit: make ***independent*** changes to a working copy

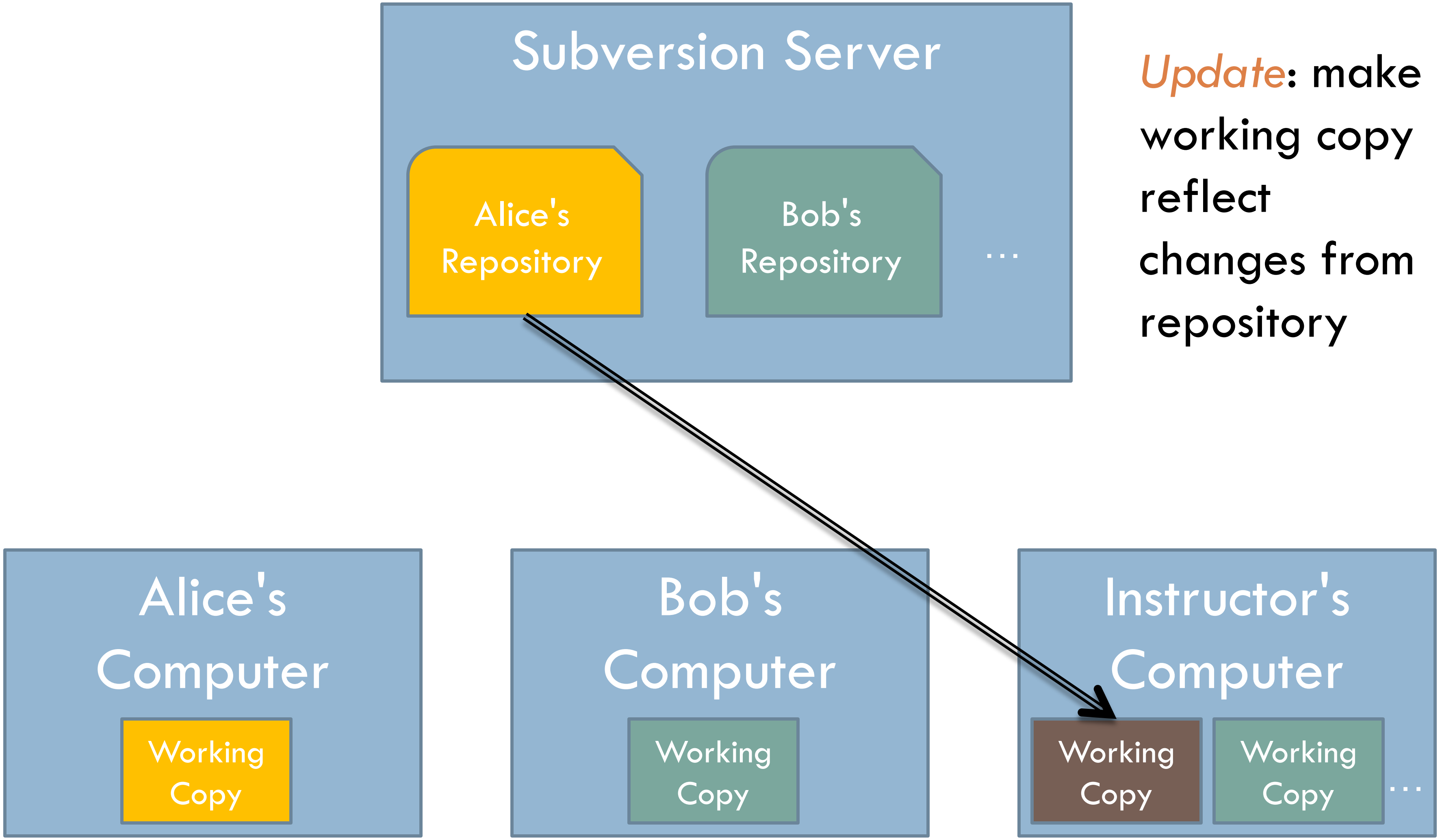
Version Control Steps—Commit



Commit: send a snapshot of changes to the repository

Q2b

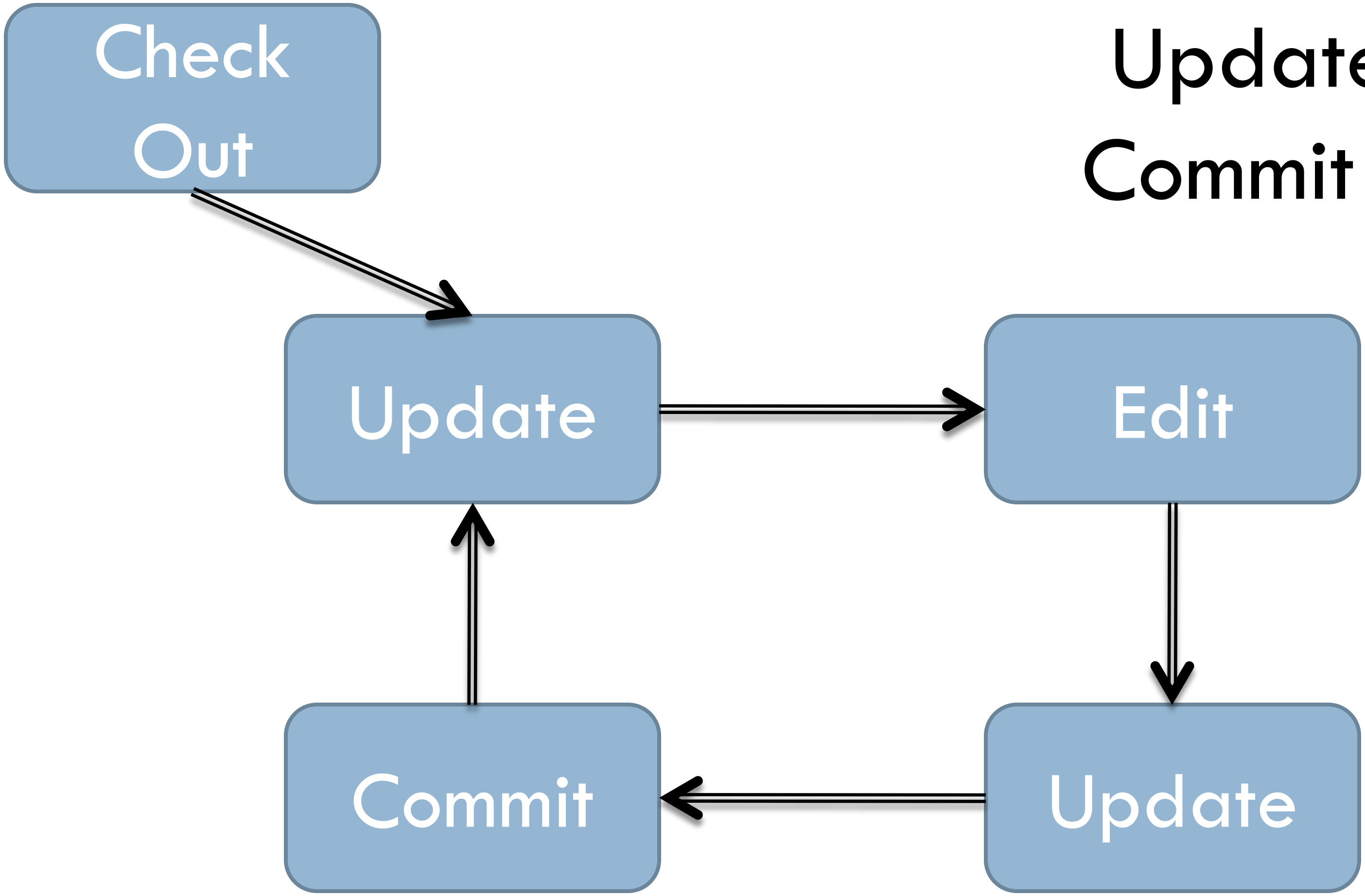
Version Control Steps—Update



Update: make working copy reflect changes from repository

Q2c

The Version Control Cycle



Update and Commit often!

If you're stuck, get help and see Step 3 of http://www.rose-hulman.edu/class/csse/csse120/201030robotics/Homework/hw05-installEclipse.html

Check out today's exercise

- Go to the SVN Repository view at the bottom of the workbench
  - If it is not there, Window→Show View→Other→SVN Repositories→OK
- Browse SVN Repository view for Session07 project
- Right-click it, and choose Checkout
  - Accept options as presented
- In Package Explorer, find sumPowers.py inside your Session07 project
- Do the first TODO (put your name on line 1), and commit your changes

Why functions?

- A function allows us to group together several statements and give them a name by which they may be invoked.
  - Abstraction (easier to remember the name than the code)
  - Compactness (avoids duplicate code)
  - Flexibility (parameters allow variation)
- Example:

```
def complain(complaint):
    print "Customer:", complaint
```

Q3

Functions in different realms

We compare the mechanisms for defining and invoking functions in three different settings:

□ Standard mathematical notation

□ Maple

□ Python

Functions in Mathematics

- Define a function:
  - $f(x) = x^2 - 5$

Formal Parameter. Used so that we have a name to use for the argument in the function's formula.

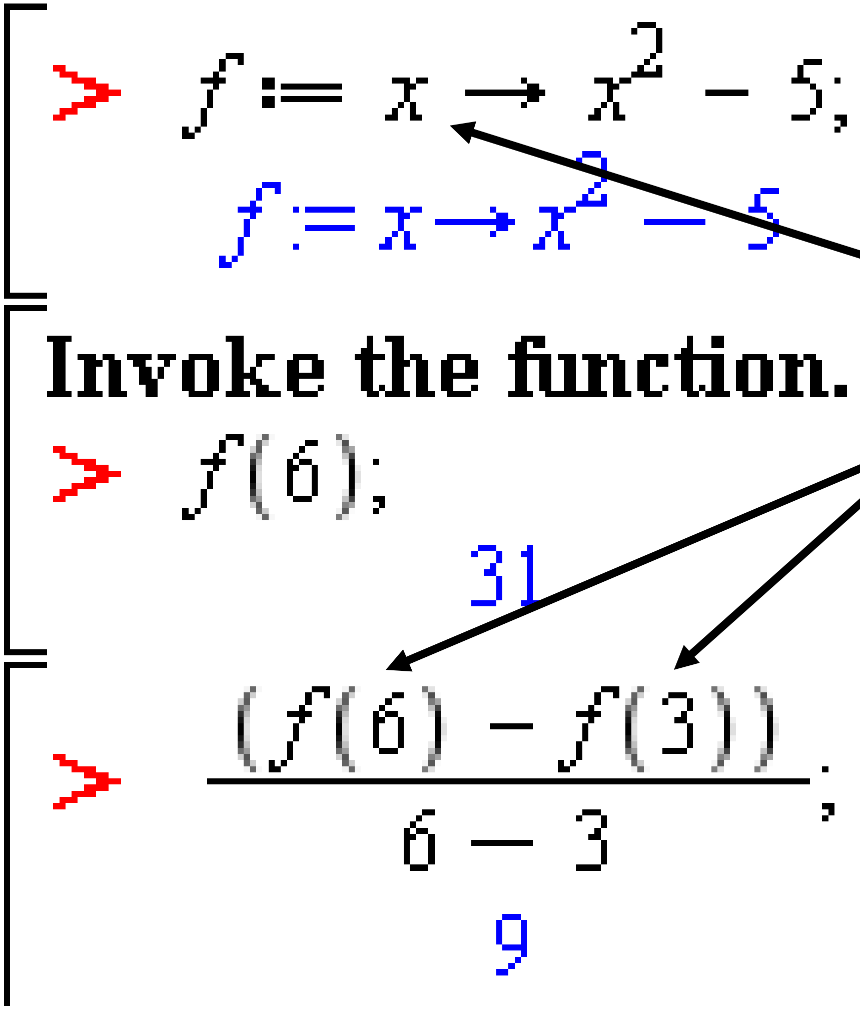
- Invoke (call) the function:
  - $$\frac{f(6) - f(3)}{6 - 3}$$

Two calls to function **f**. The first with actual parameter 6, and the second with 3.

- When the call f(6) is made, the actual parameter 6 is substituted for the formal parameter x, so that the value is $6^2 - 5$.
  - Some people use the term *actual argument*, or just *argument*, where we used *actual parameter*

Q4

Functions in Maple



> $f := x \rightarrow x^2 - 5;$

$$f := x \rightarrow x^2 - 5$$

Invoke the function.

> $f(6);$

$$31$$

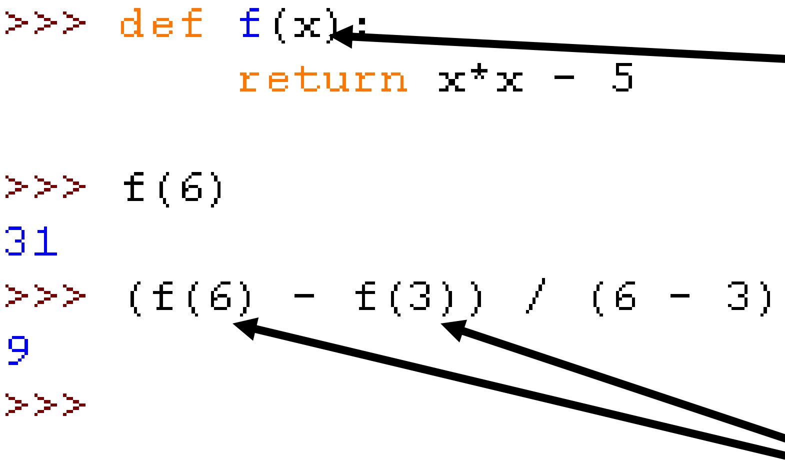
> $\dfrac{(f(6) - f(3))}{6 - 3};$

$$9$$

Formal Parameter. Used so that we have a name to use for the argument in the function's formula.

Two calls to function **f**. The first with actual parameter 6, and the second with 3.

Functions in Python

- 

```
>>> def f(x):
        return x*x - 5

>>> f(6)
31
>>> (f(6) - f(3)) / (6 - 3)
9
>>>
```

Formal Parameter. Used so that we have a name to use for the argument in the function's formula.

Two calls to function **f**. The first with actual parameter 6, and the second with 3.

  - How would you evaluate **f(f(2))**?

- In Mathematics, functions calculate a value.
- In Python we can also define functions that instead *do something*, such as print some values.

Q5

Review: Parts of a Function Definition

Defining a function called “hello”

```
>>> def hello():
      print "Hello"
      print "I'd like to complain about this parrot"
```

Blank line tells interpreter that we’re done defining the hello function

Indenting tells interpreter that these lines are part of the hello function

Review: Defining vs. Invoking

- Defining a function says what the function should do
- Invoking a function makes that happen
  - Parentheses tell interpreter to *invoke* (aka *call*) the function

```
>>> hello()
Hello
I'd like to complain about this parrot
```

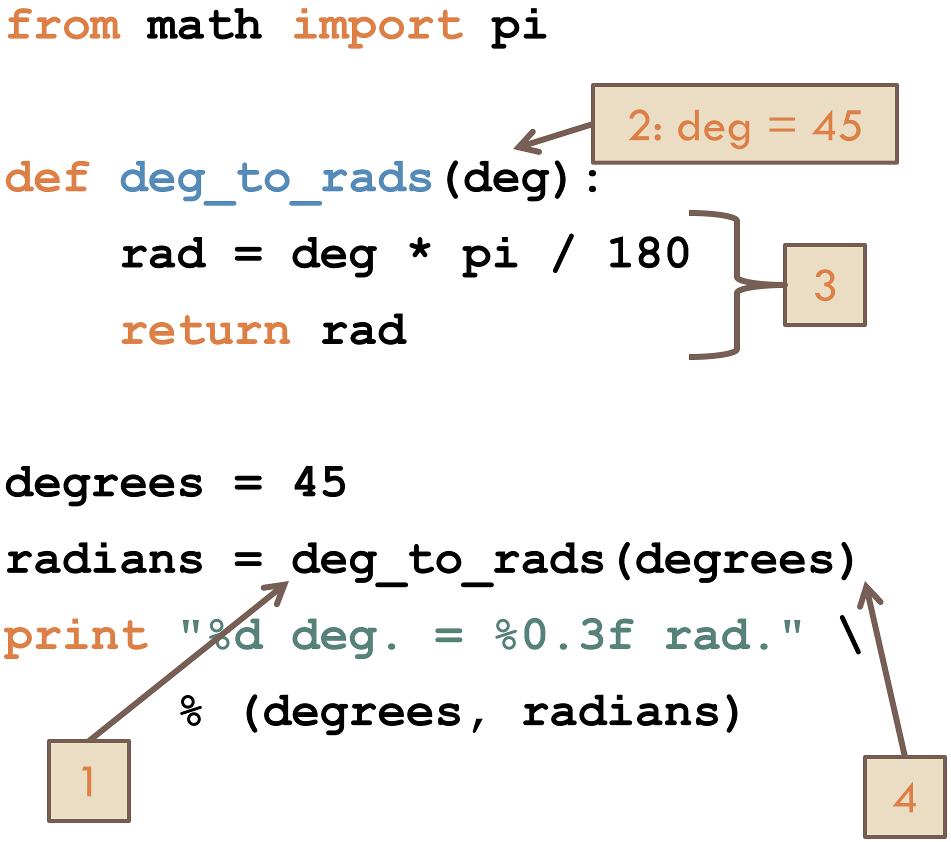
Q6

Review: Function with a Parameter

-
  ```
  def complain(complaint):
          print "Customer: I purchased this parrot not half " +
                  "an hour ago from this very boutique"
          print "Owner: Oh yes, the Norwegian Blue. " +
                 " What's wrong with it?"
          print "Customer:", complaint
  ```
- invocation:
  - complain("It's dead!")

When a function is invoked (called), Python follows a four-step process:

1. Calling program pauses at the point of the call
2. Formal parameters get assigned the values supplied by the actual parameters
3. Body of the function is executed
4. Control returns to the point in calling program just after where the function was called



```
from math import pi

def deg_to_rads(deg):
    rad = deg * pi / 180
    return rad

degrees = 45
radians = deg_to_rads(degrees)
print "%d deg. = %0.3f rad." \
      % (degrees, radians)
```

Functions can (and often should) return values

- We've **written** functions that just do things
  - `hello()`
  - `complain(complaint)`
- We've **used** functions that *return* values
  - `abs(-1)`
  - `range(10)`
- Now let's **define a function that returns a value**

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

← *return statement*

Why might it be better to **return** than **print** when a function performs a calculation?

Answer: so that we can use the returned value in expressions, e.g.

```
print square(x) + cube(x)
```

Q7

Exercise – writing a `sumPowers()` function

- Go to the sumPowers module in the Session07 project you checked out in Eclipse

- Do the TODO's
  - There are 4 TODO's
  - The last one is in *main*, near the bottom of the file

- When you believe that your *sumPowers* is correct (notice that we gave you test cases!), commit your code back to your repository

If a Function Calls a Function …

```
def g(a,b):
    print a+b, a-b

def f(x, y):
    g(x, y)
    g(x+1, y-1)

f(10, 6)
```

- Trace what happens when the last line of this code executes
- Now do the **similar** one on the quiz

Q8

An exercise in code reading

- With a partner, read and try to understand the code that is on the handout.
- You can probably guess what the output will be. But how does it work?
- Figure that out, discuss it with your partner and answer quiz question 10.
  - Optional Challenge Problem for later, just for grins: try to write "There's a Hole in the Bottom of the Sea" or "The Green Grass Grew All Around" in a similar style.
- When you are done, turn in your quiz and start the homework

Q9-10

File Processing

- Manipulating data stored on disk
- Key steps:
  - *Open* file
    - For reading or writing
    - Associates file on disk with a *file variable* in program
  - *Manipulate* file with operations on the file variable
    - *Read* or *write* information
  - *Close* file
    - Causes final “bookkeeping” to happen

Note: disks are slow, so changes to the file are often kept in a ***buffer*** in memory until we close the file or otherwise “flush” the buffer.

File Writing in Python

- Open file:
  - Syntax: <filevar> = ***open***(<name>, <mode>)
  - Example: `outFile = open('average.txt', 'w')`
    - Replaces contents!
- Write to file:
  - Syntax: <filevar>.***write***(<string>)
  - Example: `outFile.write("And this isn't my nose.\ It's a false one.")`
- Close file:
  - Syntax: <filevar>.***close***()
  - Example: `outFile.close()`

File Reading in Python

- Open file: `inFile = open('grades.txt', 'r')`
- Read file:
  - <filevar>.***read***()   Returns one **BIG** string
  - <filevar>.***readline***()   Returns next line, including \n
  - <filevar>.***readlines***()   Returns **BIG** list of strings, 1 per line
  - ***for <lineVar> in <filevar>***   Iterates over lines efficiently
- Close file: `inFile.close()`
- When you are done, start working on the homework
  - When both you and your robot partner are ready, work on the robotics problem *RobotPathViaPoints*

A “Big” Difference

- Consider:

  -
    ```
    inFile = open ('grades.txt', 'r`)
    for line in inFile.readlines():
      # process line
    inFile.close()
    ```

  -
    ```
    inFile = open ('grades.txt', 'r`)
    for line in inFile:
      # process line
    inFile.close()
    ```
- Which takes the least memory?
  - Answer: the second approach, because in it Python reads lines into memory one at a time and only as needed instead of all at once, as in the first approach