As you arrive:

1. Start up your computer and plug it in
2. ***Log into Angel*** and go to CSSE 120
3. Do the ***Attendance Widget*** – the PIN is on the board
4. Go to the course ***Schedule Page***
   - From your *bookmark*, or from the *Lessons* tab in Angel
5. Open the ***Slides*** for today if you wish

Sequences, Indexing

- Negative indices, slicing

Strings, Format specifiers

Files

- Reading, Writing, Open/Close

Functions

- Defining
- Calling (invoking)
- Parameters
- Returned values

Plus in-class time working on these concepts, continued as homework.

Session 7

CSSE 120 – Fundamentals of Software Development

Outline

- Sequences, indexing: negative indices, slicing
- Strings: Format specifiers
- Files: reading/writing, open/close, error-handling
- Functions:
  - Defining
  - Calling (invoking)
  - Sending information to a function
    - Parameters and Actual arguments
  - Getting information back from a function
    - The return expression and capturing a returned value in a variable

Plus in-class time working on these concepts, continued as homework.

Checkout today's project

- Go to SVN Repository view, at bottom of the workbench
  - If it is not there,
    Window→Show View→Other→SVN → SVN Repositories
- Browse SVN Repository view for
  07-FilesAndFunctions project
- Right-click it, and choose Checkout
  - Accept options as presented
- Expand the 07-FilesAndFunctions project that appears in Package Explorer (on the left-hand-side)
  - Browse the modules.
  - Let us explore the code in the **01-indexing.py** module

Sequences and Indexing

```
sentence = "It's just another day."
print(sentence[0])
print(sentence[
      len(sentence) - 1])
print(sentence[3:8])
print(sentence[:8])
print(sentence[8:])
print(sentence[-1])
print(sentence[-2])
print(sentence[-3:-8])
print(sentence[-8:-3])
print(sentence[:-3])
print(sentence[-3:])
print(sentence[:-1])
```

```
I
.
s jus
It's jus
t another day.
.
y

her d
It's just another d
ay.
It's just another day
```

Q1

String formatting - Example

- Allows us to *format* complex output for display
- Here's an example. Can you guess what this code prints?

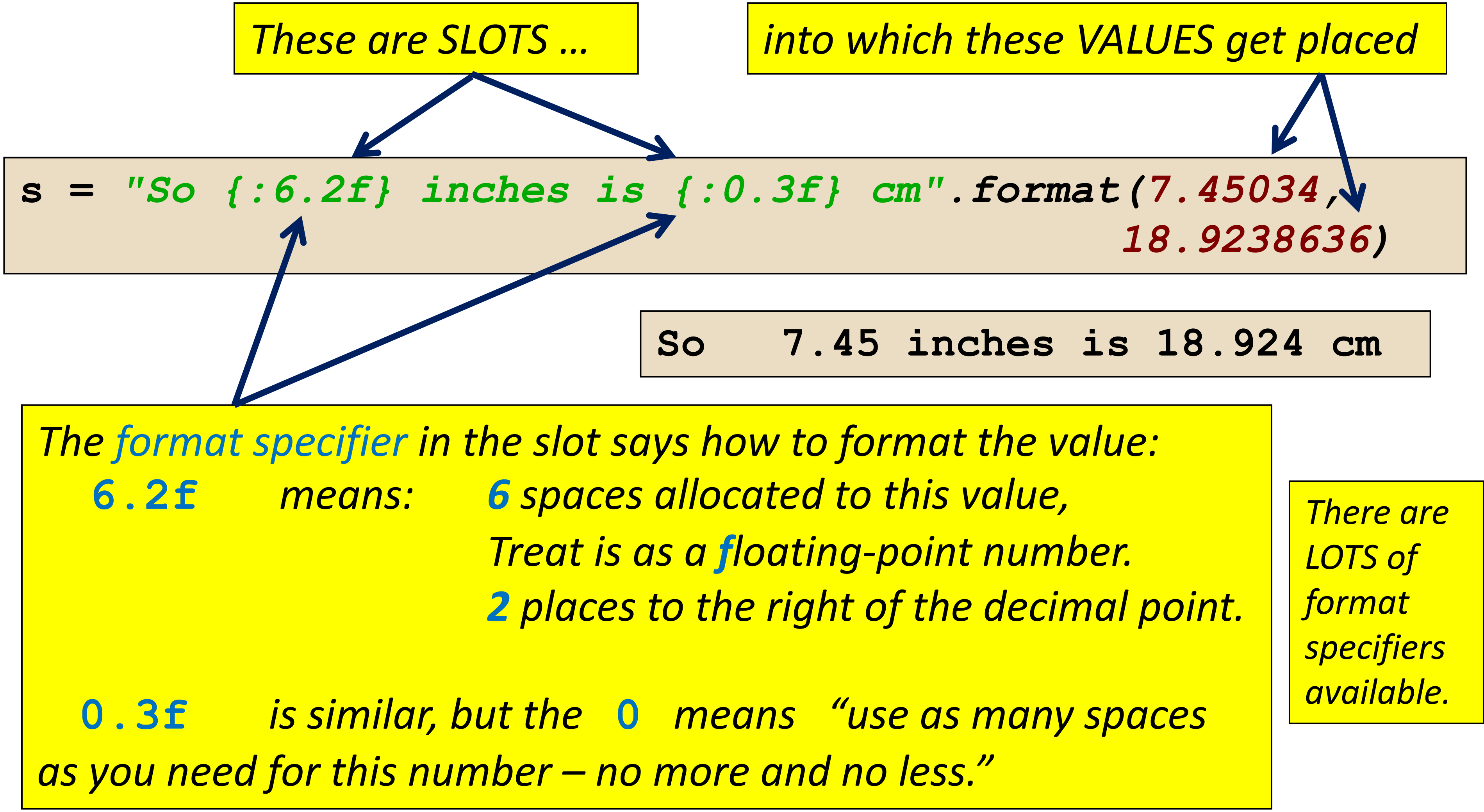
```
s = "So {:6.2f} inches is {:0.3f} cm".format(7.45034,
                                            18.9238636)
print(s)
```

- Answer:

```
So   7.45 inches is 18.924 cm
```

- The *slots* `{:6.2f}` and `{:0.3f}` get replaced by the *values* `7.45034` and `18.9238636`.
  - Left-to-right. Number of values >= number of slots (in this e.g.)
  - The *format specifier* in each slot specifies how to format its value. Details on next slide.
  - The `format` method is a built-in string method.

String formatting – Example explained



These are SLOTS …

into which these VALUES get placed

```
s = "So {:6.2f} inches is {:0.3f} cm".format(7.45034,
                                           18.9238636)
```

```
So   7.45 inches is 18.924 cm
```

The format specifier in the slot says how to format the value:

`6.2f` *means:* *6 spaces allocated to this value,*
Treat is as a floating-point number.
2 places to the right of the decimal point.

`0.3f` *is similar, but the* `0` *means “use as many spaces as you need for this number – no more and no less.”*

There are LOTS of format specifiers available.

```
s = "So {:6.2f} inches is {:0.3f} cm".format(7.45034,
                                             18.9238636)
```

String formatting – General form

General form:

<template string>`.format(`*<values>*`)`

`format` *returns* the formatted string

Recall that the angle-brackets < > are just part of the notation that we use to describe the syntax formally.

The *template string* is a string with *slots* in it, where each slot has the form:

{ : *<format specifier>* }

- Curly braces { } in the template string indicate the *slots* to be filled from the tuple of *values*.
  - If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}
- Put a ***colon*** in front of each *format specifier*.
  - You can omit the colon in **certain special** circumstances.
- Next slide: *format specifiers* for doing all sorts of things.

values is a tuple

- Each *slot* in the *template string* is filled in with the corresponding *value* in the *values* tuple (left to right).
- There must be exactly as many *slots* as values in the *values* tuple.

Format specifiers – examples

Type of data	Example format specifier	Meaning	Example result
float (but an **int** is OK and converted to a **float**)	6.2f	**Fixed point:** 2 digits after the decimal point in a field of at least 6 characters. Fill with spaces as needed. Round (using round-to-nearest-*even*-integer for ties).	45.935 → " 45.94"
	6.2e	**Exponent notation:** scientific notation. 6.	4.08e+22
	6.2g	**General format:** fixed point unless too big, then exponent.	
int	7	**Decimal:** base 10, use a field of at least 7 characters. Fill with spaces as needed. Same as **7d** (the d is the default).	45 → " 45"
	,7	Ditto, but use a **comma for thousand's separators.**	" 4,503"
	7b	Same as **7d**, but in ***binary***. Likewise: **o** for *octal*, **x** for *hexidecimal*, **c** for ***character*** (*int* converted to its Unicode).	203 → 11001011
	07	**Pad with leading 0's.** Works for *float*'s too.	"0000045"
Any type	>6	**Right-align,** use a field of at least 6 characters. Fill with spaces.	" bob"
	<6	Ditto, but **left-align.** Default is *right* for numbers, *left* for all else.	"bob "
	^6	Ditto, but **center.**	" bob "
	*^6	Ditto, but fill with *'s. Can be any character, any alignment.	"*bob**"
Non-numeric	5.2	Use at most 2 characters from the data item. Then, use a field of at least 7 characters.	"cute"→ "cu "

String formatting – Indexing e.g.

- Here's an example that uses indexing. Can you guess what this code prints?

These are indices … from this tuple

```
s = "So {2:6.2f} inches is {3:0.3f} cm".format(7.45034,
                                 18.9238636, 100.0, 254.0)
print(s)
```

- Answer:

```
So 100.00 inches is 254.000 cm
```

- The *slots* `{2:6.2f}` and `{3:0.3f}` get replaced by the *values* *`100.0`* and *`254.0`*.
  - Index into tuple. Number of values can be less than, greater than, or equal to number of slots (in this e.g.)
  - The *format specifier* in each slot specifies how to format its value. Details on next slide. NOTE the use of indexing

Format specifiers – Gory details

□ Syntax: A *format specifier* has the form:

```
[ [ fill ] align ] [ sign ] [#] [0] [ width ] [ , ] [ .precision] [type]
```

where

```
fill  ::= <a character other than '}'>
align  ::= "<"  |  ">"  |  "="  |  "^"
sign  ::= "+"  |  "-"  |  " "
width  ::= integer
precision  ::= integer
type  ::= "b"  |  "c"  |  "d"  |  "e"
        |  "E"  |  "f"  |  "F"  |  "g"
        |  "G"  |  "n"  |  "o"  |  "s"
        |  "x"  |  "X"  |  "%"
```

Briefly, this means:

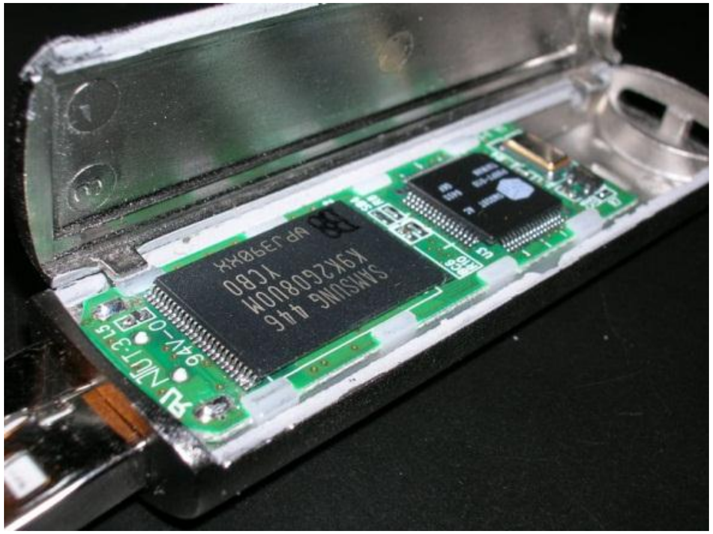
- optional stuff (including ways to left-align, right-align or center, as well as to specify a character with which to fill (pad))
- then *width.precision*, where *width* specifies that the field will be at least that wide, and *precision* gives the number of digits past the decimal point (for numbers) or the maximum number of characters to use from the data (for non-numeric data)
- then *type character*, usually *f* (for fixed point numeric) or blank.

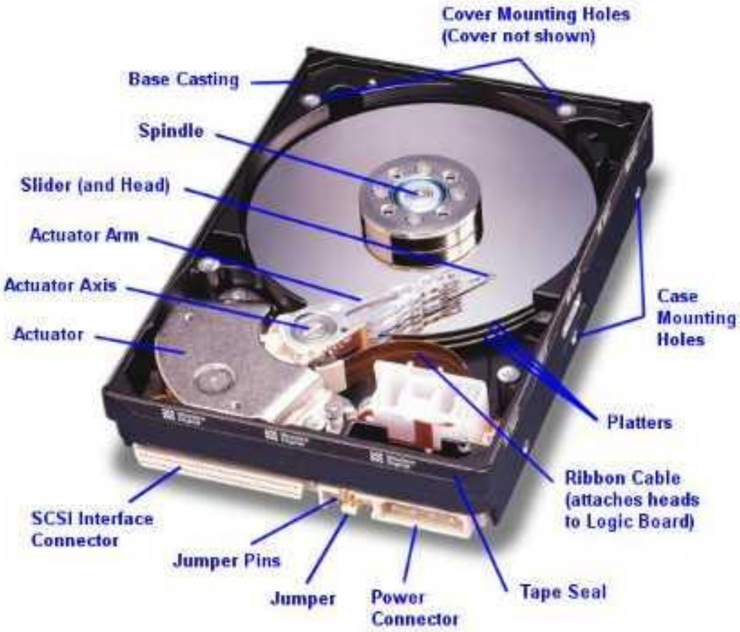
For all the gory details, see: http://docs.python.org/py3k/library/string.html#formatstrings

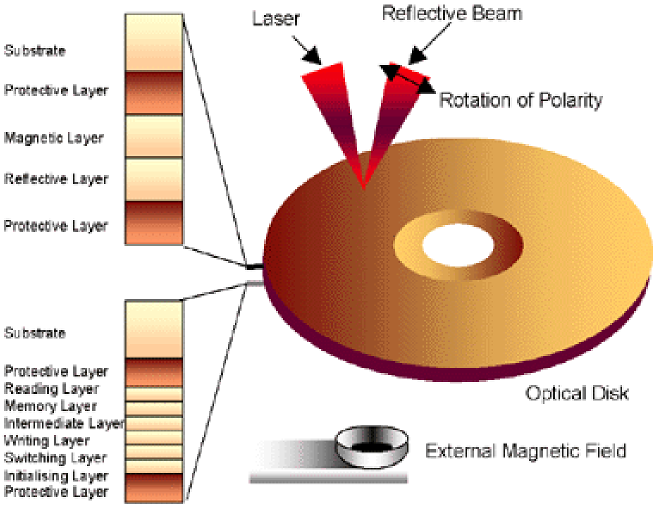
File Processing – What is a File?

- From Wikipedia: A computer file is
  - a *block* of arbitrary information,
  - or a *resource* for storing information,
  - which is available to a computer program
  - and is usually based on some kind of *durable* storage.
    - A file is *durable* in the sense that:
      - it remains available for programs to use after the current program has finished, and
      - persists even after the computer is turned off (i.e. is *non-volatile* – does not require power to maintain the stored information).
    - Computer files can be considered as the modern counterpart of paper documents which traditionally are kept in offices' and libraries' files, and this is the source of the term.

File Processing – Devices







File Processing

- Key steps:
  - *Open* file
    - For reading or writing
    - Associates file on disk with a *file variable* in program
    - Raises an *IOError* if it cannot open the file
      - xxx
  - *Manipulate* file with operations on the file variable
    - *Read* or *write* information
  - *Close* file
    - Causes final “bookkeeping” to happen
    - The devices on which files are stored are slow (compared to *main memory*), so changes to the file are often kept in a ***buffer*** in memory until we close the file or otherwise “flush” the buffer.

Q2

File ***Writing*** in Python

Operation	Syntax, then an Example
Open the file for writing	*<file variable>* `= open(`*<file name>*`, `*<mode>*`)`
	`outFile = open('average.txt', 'w')`
Write to the file	*<file variable>*`.write(`*<string>*`)`

	`s = ...`
	`outFile.write(s)`

Close the file	*<file variable>*`.close()`
	`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 <ind> in <filevar>	Iterates over lines efficiently

- Close file: inFile.close()

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?

Q3

Your turn

- Implement the following functions as described in the `03-files.py` module in today's `07-FilesAndFunctions` project
  - `writeStuffToFile()`
  - `readAndPrintMyself()`

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 / Power (parameters allow variation)
- Example:

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

Q4

Review: Parts of a Function Definition

Defining a function called “hello”

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

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

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

Review: Defining vs. Invoking

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

```
hello()
```

```
Hello
I'd like to complain about this parrot
```

Q5

Review: Function with a Parameter

Parameter, *information that comes INTO the function. Use the parameter in the body of the function.*

- □ Definition:

```
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)
```

Parameter *being used in the body of the function.*

- □ Invocation: `complain("It's dead!")`

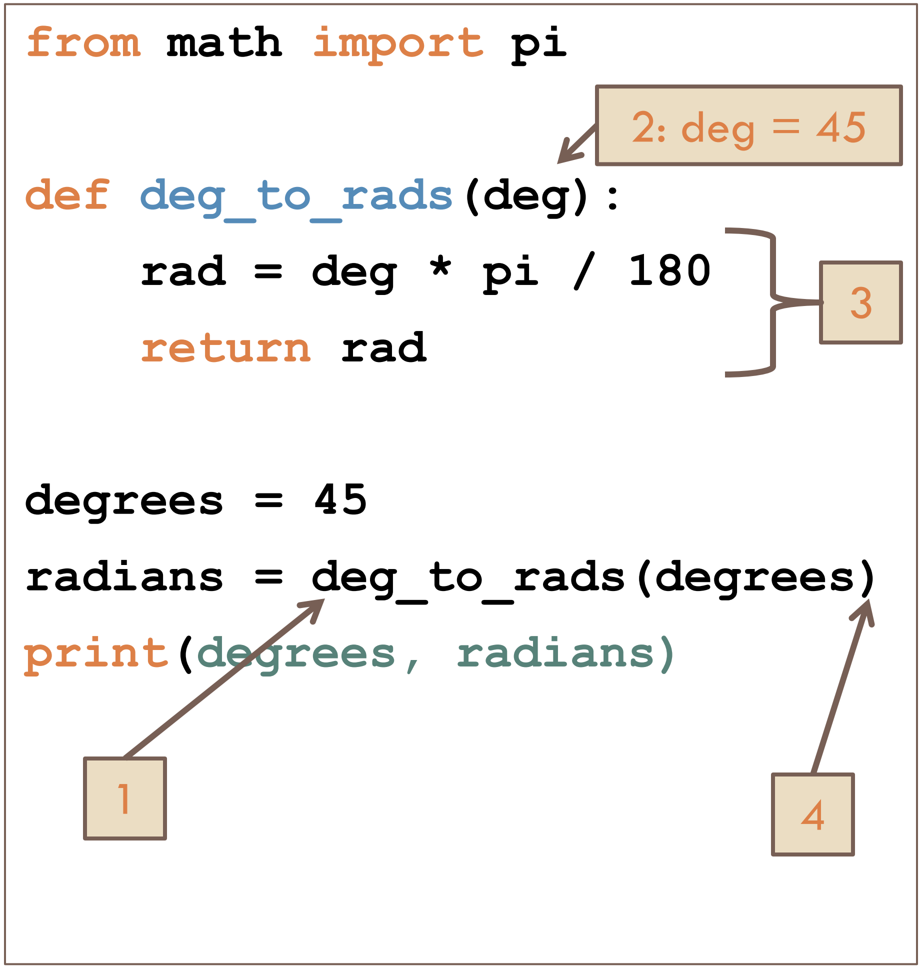
Actual argument: *the parameter is set to this value when this invocation of the function executes*

- □ Prints:

```
Customer: I purchased this parrot not
half an hour ago from this very boutique
Owner: Oh yes, the Norwegian Blue. What's wrong with it?
Customer: 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 arguments.
3. Body of the function is executed.
   - The function may *return* a value.
4. Control returns to the point in calling program just after where the function was called.
   - If the function returned a value, we capture it in a variable or use it directly.



```python
from math import pi

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

degrees = 45
radians = deg_to_rads(degrees)
print(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)
  - fn_root_1 = math.sqrt(b*b – 4*a*c)
- 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?

Q6

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

Q7

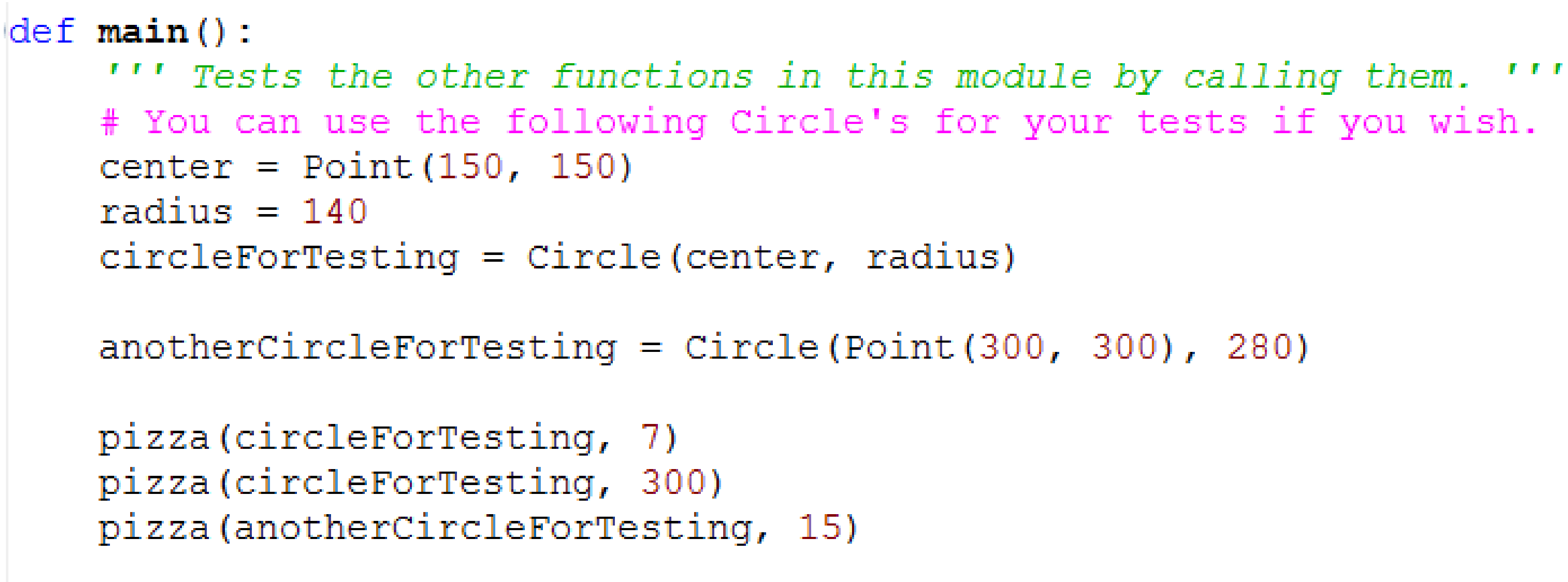
An exercise in code reading

- 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: 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 HW***

Q8-9, turn in quiz

Functions – Pizza example, ***`main`***

- □ Call **`pizza`** with ***actual arguments***.
- □ Call it several times, with different arguments. That's the *power* of parameters!

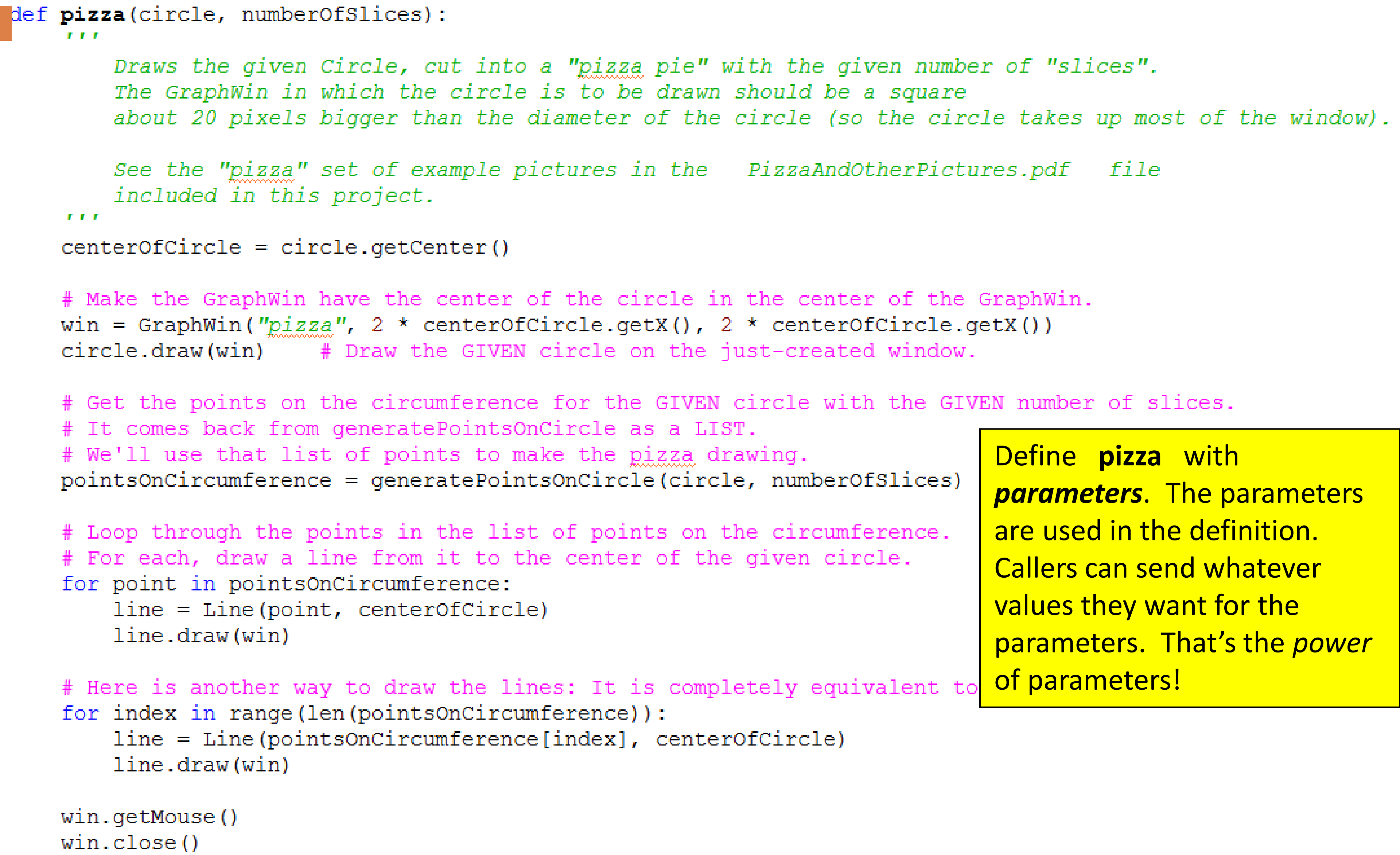


```
def main():
    ''' Tests the other functions in this module by calling them. '''
    # You can use the following Circle's for your tests if you wish.
    center = Point(150, 150)
    radius = 140
    circleForTesting = Circle(center, radius)

    anotherCircleForTesting = Circle(Point(300, 300), 280)

    pizza(circleForTesting, 7)
    pizza(circleForTesting, 300)
    pizza(anotherCircleForTesting, 15)
```

Functions – Pizza example, **pizza**



```
def pizza(circle, numberOfSlices):
    '''
        Draws the given Circle, cut into a "pizza pie" with the given number of "slices".
        The GraphWin in which the circle is to be drawn should be a square
        about 20 pixels bigger than the diameter of the circle (so the circle takes up most of the window).

        See the "pizza" set of example pictures in the   PizzaAndOtherPictures.pdf   file
        included in this project.
    '''
    centerOfCircle = circle.getCenter()

    # Make the GraphWin have the center of the circle in the center of the GraphWin.
    win = GraphWin("pizza", 2 * centerOfCircle.getX(), 2 * centerOfCircle.getX())
    circle.draw(win)    # Draw the GIVEN circle on the just-created window.

    # Get the points on the circumference for the GIVEN circle with the GIVEN number of slices.
    # It comes back from generatePointsOnCircle as a LIST.
    # We'll use that list of points to make the pizza drawing.
    pointsOnCircumference = generatePointsOnCircle(circle, numberOfSlices)

    # Loop through the points in the list of points on the circumference.
    # For each, draw a line from it to the center of the given circle.
    for point in pointsOnCircumference:
        line = Line(point, centerOfCircle)
        line.draw(win)

    # Here is another way to draw the lines: It is completely equivalent to
    for index in range(len(pointsOnCircumference)):
        line = Line(pointsOnCircumference[index], centerOfCircle)
        line.draw(win)

    win.getMouse()
    win.close()
```

Define **pizza** with ***parameters***. The parameters are used in the definition. Callers can send whatever values they want for the parameters. That's the *power* of parameters!

Rest of today

- Work on homework