Module 6: Elementary Programming in Python

Topics:

- Introduction to Imperative Programming
- Assignment Statements in Python
- Input/Output in Python
- Types of data in Python
- Conditional statements and functions in Python

Readings: ThinkP 1,2,3,5,6

Introducing Python …

- We will learn to do the things we did in Scheme
- We will learn to do new things we didn't do in Scheme
- Why change?
  - A different programming paradigm (approach)
  - More experience for you
  - Design recipe not limited to one language or style of programming!

Functional vs Imperative languages in problem solving

- With a functional language like Scheme:
  - Determine needed data types and variables
  - Determine needed functions
  - Produce a value
- With an imperative language like Python:
  - Determine needed data types and variables
  - Determine needed steps or actions
  - Keep track of how the data changes as the program executes
  - Produce a value by having an effect on the screen

Running a Python Program

- Uses an interpreter like Scheme (unlike most imperative languages)
- Most imperative languages use a compiler
  - Write entire program
  - Translate into computer-executable code
  - Run
- Generally, harder to debug with a compiler.

What does a Python program look like?

- A series of statements
  - Assignment statements
  - Function calls
- May include function definitions
  - Made up of statements
- May include new type definitions *(Module 9)*

Some Python Basics

- Written using regular mathematical notation

```
3 + 4
5 * (3 + 4) - 1
```

- Two numeric types (integers and floating point numbers) instead of one
- Strings, booleans, lists, but not a character or symbol type

Assignment Statements
`v = expr`

- `=` is the assignment operator ("becomes")
- `v` is any variable name
- `expr` is any Python expression
- How it works:
  1. Evaluate `expr`
  2. "Assign" that value to `v`
- Assignment statements do not produce a value. They only have an effect.

A very simple Python program

```
x = 2 * (4 + 12)
y = x + 8
y
x = y * y
x
y = y - 10
y
x = "hi"
x
```


NEW!!!! Python output: printing information to the screen

```
x = 20
print x
print x+5
y = "dog"
print y
z = 42.8
print z
print x, y, z
```

More on `print`

- Does not produce a value, but has an effect
- The following statements are not valid Python:

```
x = print 42.8
print (x = 4)
```

Why?

Displaying values in Python programs

- Interactions window, for variable `x`:

```
x
print x
```

- Result *usually* looks the same, but are different
- Difference is obvious in Definitions window

➔ Need to use `print` in our programs

NEW!!!! User Input to a Python Program

```
user_input = raw_input()
```

- Program stops
- Nothing happens until the user types at keyboard
- When user hits return, a string containing all the characters before the return is produced by `raw_input`
- The string value is used to initialize the variable `user_input`
- Program continues with new value of `user_input`

More on user input

- Alternate form (preferred):

```
user_input = raw_input(prompt)
```

e.g.

```
city = raw_input("Enter hometown:")
```

- Prints the value of `prompt` before reading any characters
- Value produced by **`raw_input`** is **always** a string

Debugging your program with **`print`** statements

- If you have an error in your program, place **`print`** statements at points through out your program to display values of variables
- **IMPORTANT**: Remember to remove the **`print`** statements before submitting your code.
  - Your program may fail our tests, even if it produces the correct function values!!!

Scheme vs Python: Numeric types

- Numeric calculations in Scheme were exact, unless involving irrational numbers
  - no real difference between `3` and `3.0`
- Integers in Python are stored exactly, but other numbers are approximated by floating point values
  - **`3`** is of type **`int`**, but **`3.0`** is of type **`float`**

Scheme vs Python: Numeric types

	Scheme		Python	
Value	*Representation*	*Type*	*Representation*	*Type*
nat	exact	**nat**	exact	**int[>=0]**
int	exact	**int**	exact	**int**
rational	exact	**num**	inexact	**float**
Irrational	inexact	**num**	inexact	**float**

Recall, in Scheme:

- **`check-expect`** for testing exact values
- **`check-within`** for testing inexact values

Scheme vs Python: Numeric types (con't)

- Approximations are made at intermediate steps of calculations ➔ Round-off error

- Warning: Do not compare two floating point numbers for exact equality *(more later …)*
- Use **`int`**, **`float`**, or **`(union int float)`** in contracts, as needed

Basic Mathematical Operations

- Addition (+), Subtraction (-), Multiplication (*):
  - If combining two **`int`** values, the result is an **`int`**
  - If combining two **`float`** values, or a **`float`** and an **`int`**, the result is a **`float`**
- Division: **`x / y`**
  - If **`x`** or **`y`** is a **`float`**, the result is a **`float`**
    - This is floating point division
  - If **`x`** and **`y`** are both **`int`**, the result is an **`int`**
    - This is the quotient operation
  - Be careful!!!

Other Mathematical Operations

- Remainder: **`x % y`**
  - **`x`** and **`y`** should both be **`int`**
  - produces the **`int`** remainder when **`x`** divided by **`y`**
- Exponents: **`x ** y`**
  - **`(union int float) (union int float) -> (union int float)`**
  - produces **`x`** raised to the power of **`y`**

More useful things to know

- Python precedence operations are standard math precedence rules (BEDMAS)
- Use ## for comments (from beginning or middle of line)
- Do not use dash in variable names
  - Use underscore instead

Calling functions in Python

`fn_name (arg1, arg2, …, argN)`

- built-in function or a user-defined **`fn_name`**
- must have correct number of arguments
- separate arguments by single comma
- examples:

```
abs(-3.8) => 3.8
len("Hello There") => 11
type(5) => <type 'int'>
```

The **math** Module

- A Python module is a way to group together information, including a set of functions
- The **math** module includes constants and functions for basic mathematical calculations
- To use functions from **math**
  - Import the **math** module into your program
  - Use **math.fn** or **math.const** to reference the function or constant you want


```
import math
print math.sqrt(25)
print math.log(32,2)
print math.log(32.0, 10)
print math.floor(math.log(32.0,
  math.e))
print math.factorial(10)
print math.cos(math.pi)
print sqrt(100.3)
```

Error!! Must use `math.sqrt(100.3)`

More **math** functions

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__',
  'acos', 'acosh', 'asin', 'asinh',
  'atan', 'atan2', 'atanh', 'ceil',
  'copysign', 'cos', 'cosh', 'degrees',
  'e', 'exp', 'fabs', 'factorial',
  'floor', 'fmod', 'frexp', 'fsum',
  'hypot', 'isinf', 'isnan', 'ldexp',
  'log', 'log10', 'log1p', 'modf', 'pi',
  'pow', 'radians', 'sin', 'sinh', 'sqrt',
  'tan', 'tanh', 'trunc']
```

Getting more information about a **`math`** function

```
>>> import math
>>> print math.floor._ _doc_ _
floor(x)

Return the floor of x as a float.
This is the largest integral value
 <= x.
```


Creating new functions in Python

```
def fname (p1, p2, …, pN):
  statement1
  statement2
  …
  statementK
```

Notes:

- Indent each statement the same amount
- For function to return a value, include
  `return answer`

Example: Write a Python function that consumes 3 different integers and produces the middle value.

```
# middle: int int int -> int
# Produces the middle value of a,b,c,
# where a,b,c are all different
# Example: middle(2,8,4) => 4
# middle(4,3,0) => 3
def middle(a,b,c):
    largest = max(a,b,c)
    smallest = min(a,b,c)
    mid = (a+b+c) - largest - smallest
    return mid
```

Example: Write a Python function to compute the area of a circle with positive radius `r`

```
import math
# area_circle: float[>=0]->float[>=0]
# produces the area of a circle
# with the given radius
# Examples: area_circle(0.0) => 0.0
# area_circle(1.0) => 3.14159265
def area_circle (radius):
   return math.pi * radius * radius
```


Picky, picky, picky … Indentation in Python

Consider the function **`print_both`**:

```
## print_both: any any -> None
## Produces None
## Effects: displays the values of x and
## y, on two separate lines
## Example: print_both(3, "hi") prints 3
## on one line, hi on the next
def print_both(x,y):
  print x
  print y
```

A different indentation for **`print y`** leads to errors!

Design Recipe: Testing in Python

- Our Python functions must still be tested
- Choosing test cases will be similar to before
  - Black box tests
  - White box test
- The mechanics of testing in Python will be different (but similar) as Python does not have built-in **`check-expect`** or **`check-within`**

Testing Python functions: an Introduction

- Python functions can get information from the user in different ways:
  - Parameters
  - User input using `raw_input`
- Python functions can produce or display results in different ways:
  - Value produced using `return`
  - Data displayed on the screen using `print`

CS116 "**check**" Module

- Download the file: **check.py** from the CS116 web pages. Put a copy in the same directory as your assignment **.py** files.
- Add the following line to each assignment file:

```
import check
```

- You do NOT need to submit **check.py** when you submit your assignment files.
- A message is printed only if your test fails.

check.expect

```
## Question 3, Test 1: description
check.expect(
    "Q3T1",
    expr,
    value_expected)
```

- This function performs the test:

```
expr == value_expected
```

- Use for checking exact values (integer or non-numeric).

check.within

```
## Question 4, Test 2: description
check.within(
     "Q4T2",
     expr,
     value_expected,
     tolerance)
```

- This function performs the test:
  `abs(expr - value_expected) <= tolerance`
- Use for checking inexact values (floating point numbers in Python).

Testing: Simplest Case

- No mutation, no user input, no user output
  - Use **`check.expect`** and **`check.within`** as described on the previous slides
- Set values of parameters in the function call
- Compare the produced result to the expected result

Testing `middle`

```
## Test 1: middle is first parameter
check.expect(
  "Q1T1",
  middle(3,10,1),
  3)

## Q1, Test 2: middle is middle parameter
check.expect(
  "Q1T2",
  middle(2,5,9),
  5)
```

Testing `area_circle`

`area_circle` produces a floating point

→Don't test for exact equality

```
## Q3, Test 4: zero radius
check.within(
  "Q3T4", area_circle(0.0), 0, 0.00001)

## Q3, Test 5: positive radius (1.0)
check.within("Q3T5", area_circle(1.0),
  3.14159, 0.00001)
```


Testing Screen Output

- Give a description of expected screen output:

```
check.set_screen(
     "I like testing 3x, once per line")
```

- Call appropriate **`check`** function to test value produced by the function.
- Test will print screen output produced above and your description of what the screen output should be below for YOU to compare.

Example: Screen Output Only

```
import check
# print_it_three_times: str -> None
# Purpose: produces None
# Effects: Prints the string s three times,
# once per line
# Example: print_it_three_times("a") prints
#a
#a
#a
def print_it_three_times(s):
    print s
    print s
    print s

# Q6 Test 1: longer string - "I like testing"
check.set_screen("I like testing 3x, once per line")
check.expect("QT1", print_it_three_times("I like testing"),None)
```

There is no **`return`**, so function produces **`None`**. This value is passed to **`check.expect`**

Test Output

```
QT1 (expected screen output):
I like testing 3x, once per line

QT1 (actual screen output):
I like testing
I like testing
I like testing

-----
```

You must examine your output to see if it matches what you expected.

Note: No error message printed by `check.expect`, so `None` was correctly returned by our function.

Printing vs Returning

In Scheme, most of our functions (other than mutation-only ones) produced a value. This will not be the case in Python.

Complete the design recipe for **`f1`** and **`f2`**.

```
def f1(x):
    print x+1
def f2(x):
    return x+1
```


Testing With User Input

- Set the user inputs needed for this test in order
- Always use strings for the input values

```
check.set_input(["2","7"])
```

- Call appropriate **`check`** function for produced value
- Test function will automatically use these values (in order) when a value is expected from **`raw_input`**
- You will be warned if the list contains too few or too many values

Recall: `raw_input` produces a `str` – what if we want a number?

- Python provides two built-in casting functions to convert from a `str` to an `int` or `float`
- Examples:
  - `int("42") => 42`
  - `int("-3001") => -3001`
  - `float("5.678") => 5.678`
  - `float("-12") => -12.0`

Example: Test with User Input

```
import check

def add_two_inputs():
    # Use int(...) to convert string to int
    x = int(raw_input("Enter 1nd integer: "))
    y = int(raw_input("Enter 2nd integer: "))
    return x+y

# Test 1: two positive numbers
check.set_input(["2","7"])
check.expect("AddT1", add_two_inputs(), 9)
```


Using local variables in Python

Consider a slightly different implementation of `area_circle`:

```
import math
def area_circle (radius):
   r2 = radius * radius
   area = math.pi * r2
   return area
```

The local variables `r2` and `area` can only be used inside the function body

More on local variables

- A variable first used inside a function only exists in that function
- If your function calls a helper function, the helper function cannot access the caller's variables
- We will not declare local functions in Python (though it can be done)
- Must provide contract, purpose and effects for local helper functions

Local changes are local for "basic" parameter types

```
## increase_grade: int -> None
## Purpose: Produces none
## Effects: Prints out a grade, and
## then prints out grade+1
def increase_grade(grade):
     print grade
     grade = grade + 1
     print grade
>> my_grade = 98
>> increase_grade(my_grade)
>> print my_grade
```


More on Basic Types in Python

- The differences between integers and floating point numbers can complicate calculations
- Python has many built-in conversion functions from one basic type to another

Beware of integer division!

Note that: $(1 + 1/n)^n \rightarrow e$, as $n \rightarrow \infty$

```
## estimate_e: float[>0]-> float
def estimate_e (n):
    return (1+1/n) ** n

## Python's estimate of e: 2.718281828459045
estimate_e(100.0)          => 2.70481382942
estimate_e(1000.0)         => 2.71692393224
estimate_e(10000.0)        => 2.71814592682
estimate_e(100000)         => 1
estimate_e(1000000.0)      => 2.7182804691
```


What went wrong and how do we fix it?

Look carefully at the calculation:

`(1+1/n)**n`

- How is this calculation different if `n` is a **`float`** compared to an **`int`**?
- We need **`1/n`** to be the "real" division. How?
- Note: integer value of **`n`** violated contract!!
- Warning: Be very careful with division in Python. Be sure your types are correct!!!

How to get the type we want: More Casting and Conversion Functions

- **`float: int → float`**
  - **`float(1) => 1.0, float(10) => 10.0`**
- **`float: str → float`**
  - **`float("34.1") => 34.1,`**
  - **`float("2.7.2") => Error`**
  - **`float("23") => 23.0`**

More Casting Functions

- **`int: (union float str)→ int`**
  - **`int(4.7) => 4, int(3.0/4) => 0, int(-12.4) => -12`**
  - This is a truncation operation (not rounding)
  - **`int("23") => 23`**
  - **`int("2.3") => Error`**

- **`str: (union int float) → str`**
  - **`str(3) => "3", str(42.9) => "42.9"`**

Making decisions in Python

As in Scheme, in Python we
- Have a boolean type
- Can compare two values
- Can combine comparisons using and, or, not
- Have a conditional statement for choosing different actions depending on values of data

Comparisons in Python

- Built-in type **`bool`**:
  - **`True`**, **`False`**

- Equality testing: **==**
  - Use for all atomic values
- Inequality testing: **<, <=, >, >=**
- **!=** is shorthand for not equal

Combining boolean expressions

- Very similar to Scheme
  - `v1 and v2`
    `True` only if both `v1, v2 True`
  - `v1 or v2`
    `False` only if both `v1, v2 False`
  - `not v`
    `True` if `v` is `False`, otherwise `False`
- What's the value of

```
(2<=4) and ((4>5) or (5<4) or not(3==2))
```


Evaluating Boolean expressions

- Like Scheme, Python uses Short-Circuit evaluation
  - Evaluate from left to right, using precedence
    `not, and, or`
  - Stop evaluating as soon as answer is known
    - `or`: stop when one argument evaluates to `True`
    - `and`: stop when one argument evaluates to `False`
- `1 < 0 and (1/0) > 1`
- `1 > 0 or kjlkjjaq`
- `True or &32-_-!`

Basic Conditional Statement

```
if test:
   true_action_1
   …
   true_action_K
```

```
def test_for_2(x):
  if x==2:
     print x
     x = 3
     print x
```

Another Conditional Statement

```
if test:
   true_action_1
   …
   true_action_Kt
else:
  false_action_1
  …
  false_action_Kf
```

```
## Produces cost of
## ticket, given
## buyer's age
def ticket_cost(age):
  if age < 18:
     cost = 5.50
  else:
     cost = 9.25
  return cost
```


"Chained" Conditional Statement

```
if test1:
  action1_block
elif test2:
  action2_block
elif test3:
  action3_block
…
else:
  else_action_block
```

```
def ticket_cost(age):
  if age < 3:
     cost = 0.0
  elif age < 18:
     cost = 5.50
  elif age < 65:
     cost = 9.25
  else:
     cost = 8.00
  return cost
```


Conditional statements can be nested

```
def test_x(x):
    if x < 10:
        if x>5:
            print "small"
        else:
            print "very small"
    else:
        print "big"
```

Python so far

- Our Python coverage is now comparable to the material from the first half of CS115 (without structures and lists)
- Much more to come, but we can now write recursive functions on numbers

"Countdown" Template in Python

```
def countdown_fn(n):
  if n==0:
     base_action
  else:
     … countdown_fn(n-1) …
```


Revisiting factorial

```
## factorial: int[>=0] -> int[>=1]
## produces the product of all the
## integers from 1 to n
## example:  factorial(5) => 120
## factorial(0) => 1
def factorial (x):
    if x == 0:
        return 1
    else:
        return x * factorial( x - 1)
```

Important to include **return** statement in both base and recursive cases!

Question: What is the run-time of ***factorial(n)?***

Some limitations to recursion

```
factorial(1000) →
RuntimeError: maximum recursion
  depth exceeded
```

- There is a limit to how much recursion Python "can remember"
- Recursion isn't as common in Python as in Scheme
- Still fine for small problem sizes
- We'll see a new approach for bigger problems.

Example

Write the Python function **`n_times`** that reads an integer **`n`** from the user via the keyboard, and prints out **`n`** once per line on **`n`** lines.

Continuing a Python statement over multiple lines

- Don't finish a line in the middle of a statement!
- Python expects each line of code to be an entire statement
  - Can be a problem e.g. due to indentation
- If a statement is not done, use a \ (backslash) character to show it continues on next line
  - Not needed if you have an open bracket on the unfinished line

We are now Python programmers

- We will continue to use the design recipe
  - Must change some of our terminology
  - New format for testing
- Functions
  - Can have multiple statements
  - Order of statements critical
  - Mutation common
  - Can be recursive
- Printing
  - Helpful for debugging

Goals of Module 6

- Become comfortable in Python
  - Understand the basics (types and operations)
  - Understand basic input and output
  - Understand different formats for conditional statements
  - Understand how to write recursive functions