Module 5: Elementary Programming in Python

Topics:

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

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

Saying good-bye to Scheme …

Well, not quite yet

- Assignment 04
- Exams (midterm and final)
- We will refer back to Scheme as we learn Python

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
```

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**

Use these type names in Python contracts

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
- **Do not** compare two floating point numbers for exact equality *(more later …)*
- **Do not** rely on floating point values being exact!
- 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`**

Basic Mathematical Operations

- 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 ## or # 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

Type in the interactions window

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


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

A small change in indentation will lead to error

```
# tens_digit: int[>0] -> int[>=0,<=9]
# Produces the tens digit in nnn
# Examples: tens_digit(1234) => 3
# tens_digit(4) => 0
def tens_digit(nnn):
      div10 = nnn / 10
          tens = div10 % 10
  return tens
```

WARNING!! This example contains indentation 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`**

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 displayed only if your test fails.

`check.expect`

```
## Question 1, Test 1: description
check.expect(
    "Q1T1",
    expr,
    value_expected)
```

Label the test

Actual result

Expected result

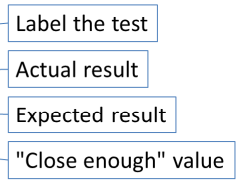
- This function performs the test:

Does **`expr`** exactly equal **`value_expected`**?

- Use for checking exact values (integer or strings).

check.within

```
## Question 2, Test 2: description
check.within(
     "Q2T2",
     expr,
     value_expected,
     tolerance)
```



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

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

```
## Q2, Test 1: zero radius
check.within(
  "Q3T4", area_circle(0.0), 0, 0.00001)

## Q2, Test 2: positive radius (1.0)
check.within("Q3T5", area_circle(1.0),
  3.14159, 0.00001)
```

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 only contract and purpose/effects for helper functions

Local changes are local for "basic" parameter types

```
## increase_grade: int -> int
## Purpose: Returns grade + 1
## Effects: None
def increase_grade(grade):
     grade = grade + 1
     return grade

>> my_grade = 98
>> increase_grade(my_grade)
>> 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[>0]
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`** are **`True`**
  - **`v1 or v2`**
    `False` only if both **`v1`**, **`v2`** are **`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 double_positive(x):
  result = x
  if x > 0:
     result = 2*x
  return result
```


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 categorize_x(x):
    if x < 10:
        if x>5:
            return "small"
        else:
            return "very small"
    else:
        return "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.

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

Example

Use recursion to write a Python function **`sum_powers`** that consumes an number (b) and a number (n) and produces the sum

$$1 + b + b^2 + b^3 + \ldots + b^n.$$

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

Goals of Module 5

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