Module 02:
Variables and Conditional Statements

Topics:
• More on Variables
• Conditional Statements
• Recursion in Python

Readings: ThinkP 5,6
Python allows us to change the values of variables

The following Python assignments are valid:

\[
\begin{align*}
x & = \ "a" \\
x & = 100 \\
x & = 2 \times x - 1
\end{align*}
\]
Can changing one variable affect another variable?

Consider running this program:

\[
x = 1000 \\
y = x \\
x = "a"
\]

What are the values of \(x\) and \(y\) now?
What does this mean for our programs?

• Values of variables may change throughout a program
• Order of execution is very important
• We can write programs that keep track of changing information, for example:
  – current location in a GPS program
  – player information in games
• We may not need a new variable for each intermediate calculation in a function
Local vs Global variables

• Variables defined inside a function are called *local* variables
  – Local variables only can be updated inside the function they are defined in

• Variables defined outside a function are called *global* variables
  – Global variables *cannot* be updated inside any functions in CS116.
Global constants

• We'll use the term *global constant* when a global variable's value is not changed after the initial assignment.

• You may use the value of any global constant inside any function you write, as you did in your Racket programs.

```python
# tax_rate = 0.13
def total_owed(amount):
    return amount * (1+tax_rate)
```
Errors with global variables

• Consider the following program:

```python
grade = 87
def increase_grade(inc):
    grade = grade + inc
>>> increase_grade(5)
```

• This causes an error. Why?

• **Do not use** `global variables` in CS116, only `global constants`.

Changing values of parameters?

Consider the program:

```python
def add1(n):
    n = n + 1
    return n
```

```python
starter = 0
>>> y = add1(starter)
• The value of \texttt{n} is changed locally, but the value of \texttt{starter} is not changed. The change to \texttt{n} is a \textit{local} change only.
• Even if \texttt{starter} was called \texttt{n}, the same behaviour would be observed.
• Note: Things are more complicated with lists. (Later...)
```
Making decisions in Python

As in Racket, in Python we

- Have a Boolean type (Bool)
- Can compare two values
- Can combine comparisons using \texttt{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 most values
  – **Never** use `==` to compare floating point values due to representation and round-off errors

• Inequality testing: `<`, `<=`, `>`, `>=`

• `!=` is shorthand for not equal
Simplify the following comparisons (assume `math` has been imported)

• $23 < 35$
• $(4 + 3 + \text{abs}(-4)) == 12$
• $5*5 > (3*3 + 4*4)$
• $5*5 \geq (3*3 + 4*4)$
• "abc" != "ABC"
• "elephant" >= "cat"
• `abs(math.sqrt(2)-1.41421)\leq 0.001`
Combining Boolean expressions

• Very similar to Racket
  – \( v_1 \) and \( v_2 \)
    True only if both \( v_1, v_2 \) are True
  – \( v_1 \) or \( v_2 \)
    False only if both \( v_1, v_2 \) are False
  – not \( v \)
    True if \( v \) is False, otherwise False

• What’s the value of
  \((2 \leq 4) \text{ and } ((4 > 5) \text{ or } (5 < 4) \text{ or not(3 == 2)))\)

• Python allows short cuts for some expressions:
  \( x_1 < x_2 < x_3 \)
Evaluating Boolean expressions

• Like Racket, 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
  – Note: an expression’s syntax is checked before the expression is evaluated. If there is a syntax error, the expression is not evaluated.

• 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

```python
if test:
    true_action_1
    ...
    true_action_Kt
else:
    false_action_1
    ...
    false_action_Kf
```

```python
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
Why are these different?

\[
x = 20
\]

\[
\text{if } x > 10: \\
\quad x = x + 1
\]

\[
\text{elif } x > 5: \\
\quad x = x - 1
\]

\[
\text{else:} \\
\quad x = 2 \times x
\]

\[
x = 20
\]

\[
\text{if } x > 10: \\
\quad x = x + 1
\]

\[
\text{if } x > 5: \\
\quad x = x - 1
\]

\[
\text{else:} \\
\quad x = 2 \times x
\]
Conditional statements can be nested

```python
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
def countdown_template(n):
    if n==0:
        return base_answer
    else:
        answer = ... n ...
        ... countdown_template(n-1) ...
        return answer
Revisiting \texttt{factorial}

def factorial (n):
    '''returns the product
    of all the integers from 1 to n
    factorial: Nat \rightarrow\ Nat
    Examples:
    factorial(5) \Rightarrow 120
    factorial(0) \Rightarrow 1
    '''
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

Important to include \texttt{return} statement in both base and recursive cases!
Some limitations to recursion

factorial(1500) ⇒
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 Racket
• Still fine for small problem sizes
• We’ll see a new approach for bigger problems.
Examples

Use recursion to write Python functions:

- **sum_powers** that consumes a positive Natural number \( b \) and a Natural number \( n \) and returns the sum
  
  \[
  1 + b + b^2 + b^3 + \ldots + b^{n-1} + b^n.
  \]

- **is_prime** that consumes a Natural number \( n \) and returns True if \( n \) is prime (its only positive divisors are 1 and \( n \)), and False otherwise.
Background: Alternate representations of boolean values

• In Python,
  – *False* and 0 are equal
  – *True* and 1 are equal
  – Any nonzero number is treated as a *True* expression in an *if* statement

• For clarity, we will continue to use *True* and *False* exclusively for our Bool values (you should follow this practice on assignments)
We are now Python programmers

• Our functions can do more ...
  – May include
    • assignment statements
    • conditional statements
    • function calls (including recursive calls)
    • `return` statements
  – Changing values of variables is common
  – Order of statements critical
Goals of Module 2

• Become comfortable in Python
  – Changing values of variables
  – Local vs global variables/constants
  – Different formats of conditional statements
  – Recursive functions