Expressions, Statements, Variables, Assignments, Types

CSE 1310 – Introduction to Computers and Programming
Vassilis Athitsos
University of Texas at Arlington

Credits: a significant part of this material has been created by Dr. Darin Brezeale and Dr. Gian Luca Mariottini

Expression

- An **expression** is a piece of code that evaluates to a value.
  - That value is called a **return value**.
- The concept of an expression is key in this course, we will refer to it many times.
- It is VERY important to differentiate what is an expression and what is not an expression.

Is it an Expression?

- 12
- 1+(2**3)
- a = 5+12
- b
- c = input("enter a number: ")
- input("enter a number: ")

Is it an Expression?

- 12 yes
- 1+(2**3) yes
- a = 5+12 no
- b yes, if b is a defined variable
- c = input("enter a number: ") no
- input("enter a number: ") yes

Is it an Expression?

- 12 yes, return value 12
- 1+(2**3) yes, return value 9
- a = 5+12 no
- b yes, if b is defined, returns value of b
- c = input("enter a number: ") no
- input("enter a number: ") yes, return value is a string of whatever we type.

How Can We Tell If *xxx* is an Expression?

- "Thinking-free way" way: type the following into Python, and see if it is accepted:

```
variable1 = (xxx)
```

- An example:
  a = 5

  To figure out if it is an expression in Python, type:

```
variable1 = (a = 5)
```

  (The result is that a = 5 is NOT an expression).

How Can We Tell If Something is an Expression?

- "Thinking" way: understand what are expressions, learn some rules.
  - No need to memorize lots of rules, just identify some basic broadly applicable rules that cover many cases.
- Example rule: Arithmetic calculations, involving numbers and variables, are expressions.

Trick Question

- Is this an expression?

  12 minus 5

Trick Question

- Is this an expression?

  12 minus 5

- Let's try answering this question the "thinking-free" way, by typing it in:

```
>>> variable1 = (12 minus 5)
```

Trick Question

- Is this an expression?

  12 minus 5

- Let's try answering this question the "thinking-free" way, by typing it in:

```
>>> variable1 = (12 minus 5)
SyntaxError: invalid syntax
```

Trick Question

- Is this an expression?

  12 minus 5

- Let's try answering this question the "thinking-free" way, by typing it in:

```
>>> variable1 = (12 minus 5)
SyntaxError: invalid syntax
```

- `12 minus 5` is not valid code (gives an error, not a result value), so it is not an expression.

Locating Expressions

- An expression can be an entire line of code.
- An expression can appear as *part* of a line of code.
- Simple expressions can be combined into complicated expressions.
  - Example: 12 + 5 + 12*9/3
    - Three subexpressions: 12, 5, 12*9/3, connected by +.
    - 12*9/3 can be further decomposed…

Locating Expressions

- Can you find expressions that are parts of lines?

```
# get the radius from the user as a string
radius_string = input("Enter the radius of your circle: ")

# convert the radius string to a real number.
radius = float(radius_string)

# compute and print the circumference
pi = 3.14159
circumference = radius * 2 * pi
print("Circumference = ", circumference)

# compute and print the area
area = (radius ** 2) * pi
print("area = ", area)
```

Locating Expressions

- Can you find expressions that are parts of lines? YES

```
# get the radius from the user as a string
radius_string = input("Enter the radius of your circle: ")

# convert the radius string to a real number.
radius = float(radius_string)

# compute and print the circumference
pi = 3.14159
circumference = radius * 2 * pi
print("Circumference = ", circumference)

# compute and print the area
area = (radius ** 2) * pi
print("area = ", area)
```

Highlighted in red you see some examples of expressions, there are more examples than the ones shown.

Remember:

-Simple expressions can be parts of more complicated expressions.

- Expressions may consist of more simple expressions.

What Does Python Do With Expressions?

- Every time Python sees an expression, it evaluates it.
- This is how Python *computes*.
- When Python evaluates a longer piece of code that includes an expression:
  - Python computes the return value of the expression.
  - Python *substitutes* the return value for the expression itself.

An Example of Expression Evaluation

```
>>> print(12+3)
```

- Here we have a line of code that includes expression 12+3.
- When Python evaluates that line of code, it computes the return value of 12+3, which is 15.
- Then, Python simplifies the code that is evaluated, by *substituting* the return value for the expression. Thus, "print(12+3)" becomes "print(15)".

Checklist

- You should now be able to:
  - Define what an expression is.
  - Be able to determine if any piece of code is an expression or not.
  - Be able to identify expressions in any line of code.
  - Be able to identify whether an expression contains more simple expressions.
  - Be able to write some expressions by yourself (TRY IT).

Statements

- Consider this line of code:

  var1 = 7 * 3 + 1

- This line is NOT an expression (although it contains expressions).
- This line of code "does something", namely it assigns a value to variable var1.
- Statements are pieces of code that are NOT expressions, but do something.

What Statements Do

- Examples of what a statement can do:
  - Assigning a value to a variable
  - Printing out a message
  - Pausing for five seconds
  - Drawing a picture
  - Playing a song
  - Downloading and displaying a web page

Statements and Expressions

- A typical program needs both expressions and statements.
- Expressions are used to *compute* new data.
- Statements are used to assign values to variables and to produce program output.

A Statement-Less Program

- Type this into a text file, and execute:

```
12+45*2
15 / 2 + 4**7
```

- Each of these two lines of code computes something (so Python will spend the time needed to compute those values).
- However, the program shows nothing to the user. Thus, the computation has been wasted.

The **print** function

- Syntax:
  print(exp_1, exp_2, …, exp_n)
  - print is followed by 0 or more expressions in parentheses, SEPARATED BY A COMMA.
  - print prints the **RETURN VALUE** of each expression.
- What will this line print?

print(12+3)

- It will print 15, it will NOT just print the text "12+3"

Expressions Can Have Side-Effects

- Consider this line:

  radius_string = input("Enter the radius of your circle: ")
- In the above line, the right side of the assignment operator contains this expression:

  input("Enter the radius of your circle: ")
- This expression also has a side effect: it prints out a message.

Assignment Statements

- An assignment statement has this syntax:

my_variable = expression

- What an assignment statement does is:
  - compute the return value of expression
  - associate that return value with the variable called my_variable.
  - from now on, my_variable is an expression whose return value is the value stored in my_variable.

The Assignment Operator

- In Python, we call the = sign the **assignment operator.**
- The assignment operator looks the same, but **is not the same as** the = sign in math.
- First difference:
  - in math, "a = 5" is the same as "5 = a".
  - In Python, "a = 5" is a valid piece of code assigning value 5 to variable a. "5 = a" is not valid code.

The Assignment Operator

- In Python, we call the = sign the **assignment operator.**
- The assignment operator looks the same, but **is not the same as** the = sign in math.
- Second difference:
  - in math, "a = a + 5" is nonsense (most of the time)
  - In Python, "a = a + 5" is a valid piece of code, and increments the value of a by 5.

The Assignment Operator

- In Python, we call the = sign the **assignment operator.**
- The assignment operator looks the same, but **is not the same as** the = sign in math.
- Third difference:
  - in math, "a + 5 = 7" is a well-defined equation.
  - In Python, "a + 5 = 7" is not a valid piece of code, because the left side of the assignment operator is NOT a variable name.

Variable Names

- Variable names must follow some simple rules:
  - must begin with a letter or an underscore.
    - DO NOT start variable names with underscores for the time being.
  - can include letters, numbers and underscores
    - but cannot start with a number.
  - are case sensitive, name and Name are different variables.
  - cannot be the same as a keyword
    - Try in Python: for = 15
    - for = 15 will be rejected by Python, because for is a reserved keyword in Python (as we will see in the next lecture or two).

Choosing Variable Names

- Python does not care what names you use (as long as you follow the rules).
- However, descriptive variable names can make a huge difference in program readability, helping both others and yourself understand and debug your code.
  - You will probably be surprised by how hard it will be for yourself to read your own code a few days or weeks after you wrote it.

Examples of Assignments

```
>>> xInt = 5
>>> yInt = 2 + 3
>>> yInt = yInt + 7
```

- NOTE: while evaluating the expression on the right side of the assignment operator, the OLD value of the variable is used.

Examples of Errors

- myInt + 5 = 7
- 7 = myInt + 5
- 8 = x

Examples of Errors

- myInt + 5 = 7
  - Left side of assignment operator is NOT a variable name.
- 7 = myInt + 5
  - Left side of assignment operator is NOT a variable name.
- 8 = x
  - Left side of assignment operator is NOT a variable name.

Operators += -= *= /=

- a += 5
  - is the same as: a = a + 5
- a -= 5
  - is the same as a = a – 5
- a *= 5
  - is the same as a = a * 5
- a /= 5
  - is the same as a = a / 5

The Notion of Syntactic Sugar

- If Python did not have +=, -=, *=, /=, it would not prevent us from writing any code.
  - We would just need to write slightly longer (but easier to read) lines.
  - The term "syntactic sugar" refers to elements of a programming language that are not vital, but simply allow somewhat shorter/more convenient alternative ways to write something.

Types

- In Python, every expression has a type.
- You can find the type of any expression using the **type** keyword

```
>>> type(4)
<class 'int'>
>>> type(10+12)
<class 'int'>
```

```
>>> a = 2.34
>>> type(a)
<class 'float'>
>>> b = "hello"
>>> type(b)
<class 'str'>
```

- For now, we care about three types:
  - integers (int), real numbers (float), strings (str)

Types

- The int type:
  - Used to store integers, like 4, 0, -10.
- The float type:
  - Used to store real numbers, like -13.34, 1.0, 10.5
- The str type:
  - Used to store *strings*, i.e., text, like:
    - "hello"
    - "today is Monday"
    - the contents of an entire book

Why Are Types Important?

- The type of an expression specifies two things:
  - the **internal structure** of the expression (what kind of data it contains)
  - the **kinds of operations** that Python can perform on that expression

int vs. str

- A simple example where types make a difference:

```
>>> 5 + 2
???
>>> '5' + '2'
???
```

int vs. str

- A simple example where types make a difference:

```
>>> 5 + 2
7
>>> ‘5’ + ‘2’
'52'
```

- Why?

int vs. str

- A simple example where types make a difference:

```
>>> 5 + 2
7
>>> ‘5’ + ‘2’
'52'
```

- In first case, 5 and 2 are ints. In second case, they are strings. Operator + is defined differently for ints and strings.

Automatic Type Assignment

- Python does not require you to pre-define the type of a variable.
  - An important difference from Java, C++.
- What type a variable holds can change.
  - Nonetheless, knowing the type can be important for using the correct operations on a variable.

An Example of Type Conversion

- From the circles.py program:

```
# get the radius from the user as a string
radius_string = input("Enter the radius of your circle: ")

# convert the radius string to a real number.
radius = float(radius_string)
```

- The **float** keyword is used to convert a string into a real number.

Type Conversion

- int(var_name) converts to an integer
- float(var_name) converts to a float
- str(var_name) converts to a string
- You should check these out:
  - int(2.1) → 2, int('2') → 2,
  - int('2.1') will fail
  - float(2) → 2.0, float('2.0') → 2.0,
  - float('2') → 2.0, float(2.0) → 2.0
  - str(2) → '2', str(2.0) → '2.0', str('a') → 'a'

The + and += Operator on Strings

```
>>> "hello" + "world"
'helloworld'
>>> a = "good"
>>> a += " morning"
>>> a
'good morning'
```

- The + operator concatenates strings together into a single string.

Strings vs. Numerical Expressions

```
>>> a = 12+3
>>> a
15
>>> a = "12+3"
>>> a
>>> 12+3
>>> print 12+3
15
>>> print "12+3"
12+3
```

do not confuse a string (which is text) looking like a numerical expression with the numerical expression itself.

Assignments Using **input** Expression

- Syntax:
  variable = input(expression)
- Such a line does the following:
  - Prints the return value of the expression (this means that the expression must be evaluated).
  - Waits for the user to write some text and press <ENTER>
  - Stores the text typed by the user into variable.

Program Execution

```
# get the radius from the user as a string
radius_string = input("Enter the radius of your circle: ")

# convert the radius string to a real number.
radius = float(radius_string)

# compute and print the circumference
pi = 3.14159
circumference = radius * 2 * pi
print("Circumference = ", circumference)

# compute and print the area
area = (radius ** 2) * pi
print("area = ", area)
```

Execute lines, starting from the first line.

For each line, **first** evaluate the expressions that are encountered on that line, **and then** execute the statements (if any) that use those expressions.

Then, figure out the next line to execute.