Introduction to Computational Thinking

Module 4 :
Variables, Data types, and Operators

Asst Prof Chi-Wing FU, Philip
Office: N4-02c-104
email: cwfu[at]ntu.edu.sg

Topics

- **Variables**
- Assignment Operator
- Data Types
- Data Conversion
- Operators
- Powerful Data Types and Random Module
- Case Study: Calculator Example

What is a Variable?

- In most computer programs, we need data storage to represent and store “something” (data) temporarily in programs

```
# 1. prompt user for the radius,
# 2. apply circumference&area formulae
# 3. print the results
import math
radiusString  = input("Enter the radius of your circle:")
radiusFloat = float(radiusString)
circumference = 2 * math.pi * radiusFloat
area = math.pi * radiusFloat * radiusFloat

print()       # print a line break
print( "The cirumference of your circle is:",circumference,\
            ", and the area is:" , area )
```

This is like M+ and MR in calculator

What is a Variable?

- We can use *names* to make our program more readable, so that the "something" is easily understood, e.g., **`radiusFloat`**

```
# 1. prompt user for the radius,
# 2. apply circumference&area formulae
# 3. print the results
import math
radiusString  = input("Enter the radius of your circle:")
radiusFloat = float(radiusString)
circumference = 2 * math.pi * radiusFloat
area = math.pi * radiusFloat * radiusFloat

print()       # print a line break
print( "The cirumference of your circle is:",circumference,\
            ", and the area is:" , area )
```

Variable **`radiusFloat`**

Variable Objects

- For each variable, Python keeps a pair of info.:
  - variable's name
  - variable's value

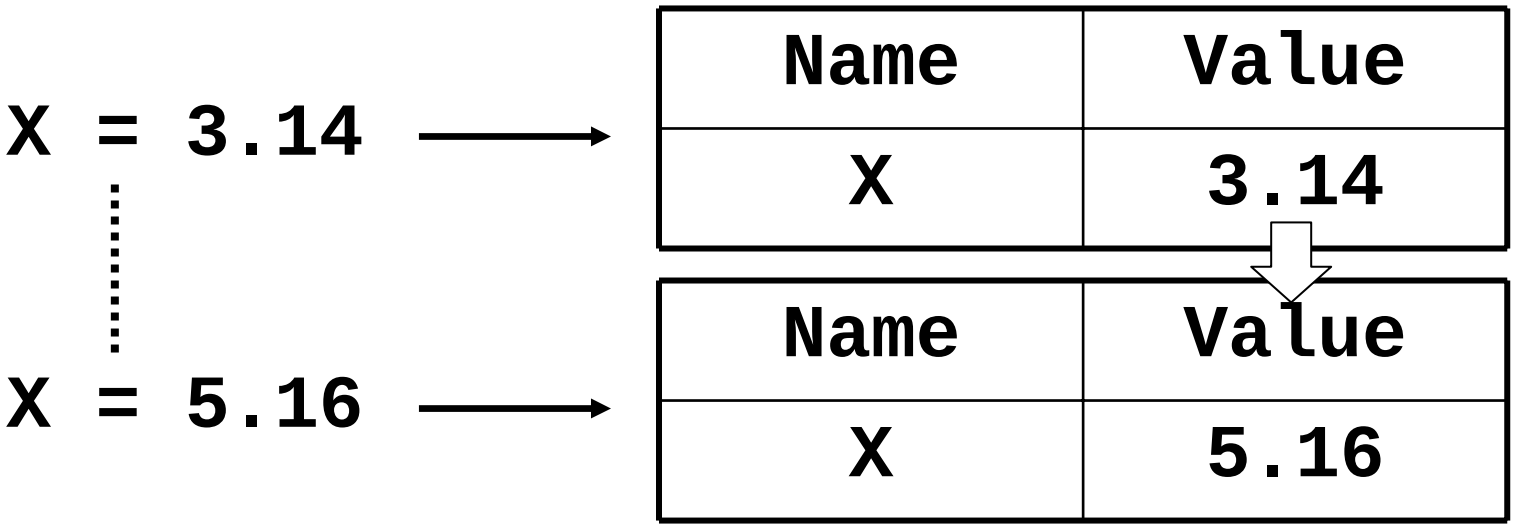
`X = 7.1` ⟶

Name	Value
X	7.1

- A variable is created when a value is assigned to it *for the first time*. It associates a name with a value.
- We say *name references value*

Variable Objects

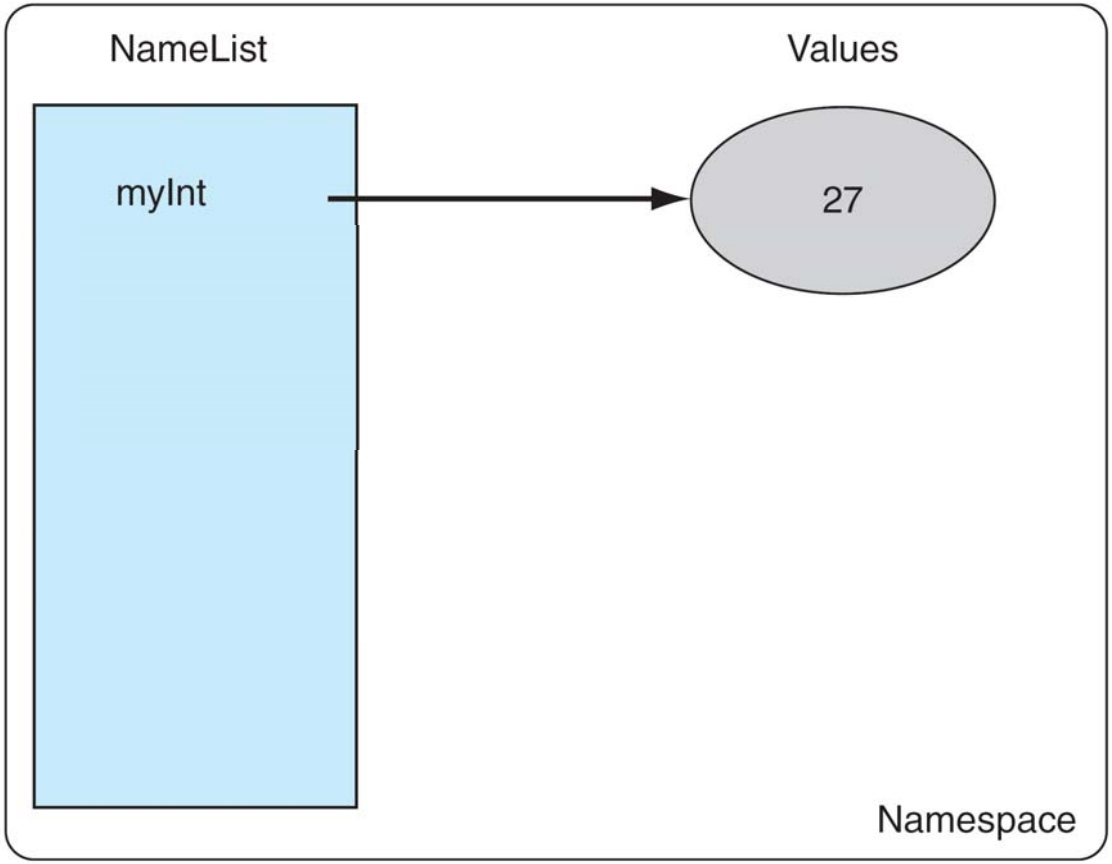
- **Operations**: Once a variable is created, we can store, retrieve, or modify the value associated with the variable name
- Subsequent assignments can update the associated value



Namespace

- A namespace is the table that contains (and keeps track of) the association of names with values
- We will see more about namespaces as we get further into Python, but it is an essential part of the language.

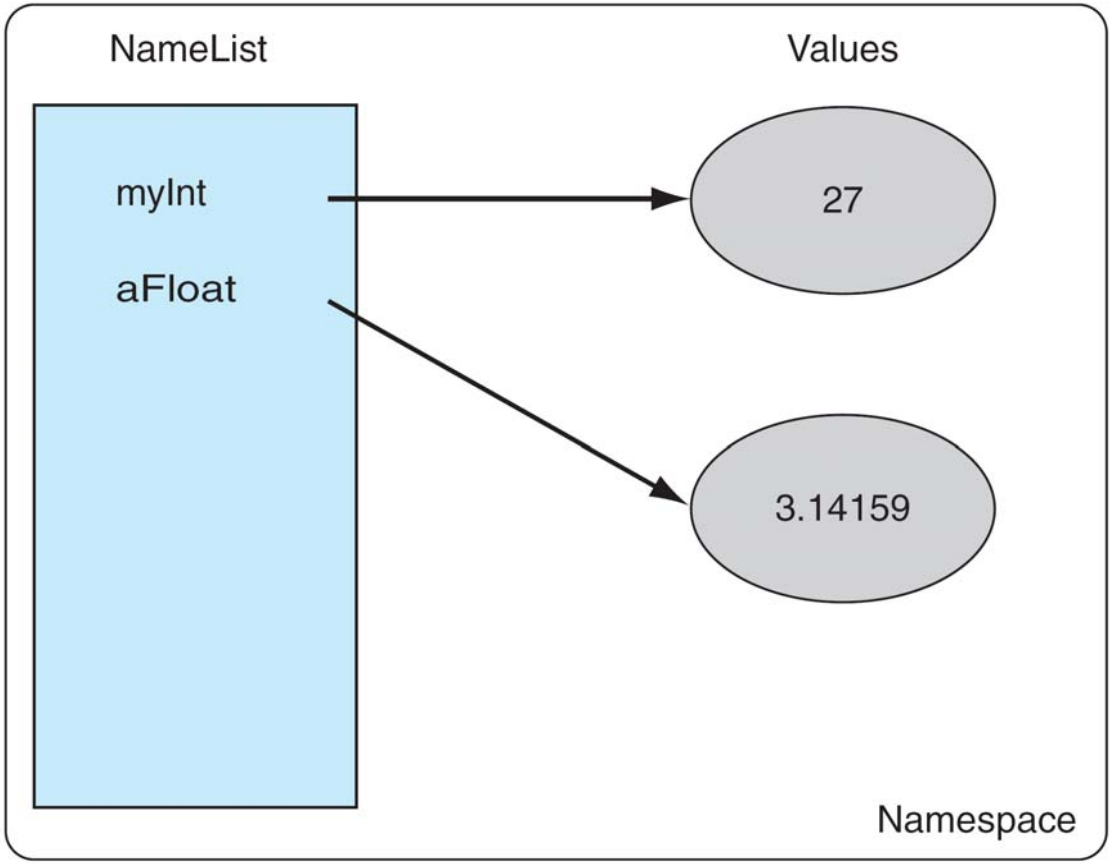
Namespace




```
>>> myInt = 27
>>> print( myInt )
27
```

FIGURE 1.1 Namespace containing variable names and associated values.

Namespace




```
>>> myInt = 27
>>> print( myInt )
27
>>> aFloat = 3.14159
>>> print( aFloat )
3.14159
```

FIGURE 1.1 Namespace containing variable names and associated values.

Python Naming Conventions

- How to name variables?

  (as well as other things that you will see later in this course, e.g., user-defined functions, etc.)

<u>One basic practice!!!</u>

Chooses the names of the variables carefully and

explains what each variable *means*

Python Naming Conventions

```
import math
radiusString  = input("Enter the radius of your circle:")
radiusFloat = float(radiusString)
circumference = 2 * math.pi * radiusFloat
area = math.pi * radiusFloat * radiusFloat
```

VS

```
import math
a  = input("Enter the radius of your circle:")
b = float(a)
c = 2 * math.pi * b
d = math.pi * b * b
```

What is c?
Not immediately clear

Same program!!!
Though both programs work…
Different names for the variables
Readability counts!!!

Python Naming Conventions

Syntax rules in Python:

- must begin with a letter or _

  `Ab123` and **`_b123`** are OK, but **`123ABC`** is not.

- may contain letters, digits, and underscores

  `this_is_an_identifier_123`

- may be of any length

- upper and lower case letters are different

  `LengthOfRope` is not **`lengthofrope`**

- names starting with _ have special meaning. Be careful!!!

Python is **case sensitive**!!

A Common Pitfall in Python

```
john_math_score  = 90
peter_math_score = 70
mary_math_score  = 80
john_eng_score   = 60
peter_eng_score  = 60
mary_eng_score   = 60

total   = john_math_score + peter_math_score + mary_math_score
average = total/3.0
print( "average Math score=" , average )
Total   = john_eng_score + peter_eng_score + mary_eng_score
average = total/3.0
print( "average English score=" , average )
```

English scores are all 60

Can we interpret and run this program?

But what's wrong?

Hint: a typo!

Topics

- Variables
- **Assignment Operator**
- Data Types
- Data Conversion
- Operators
- Powerful Data Types and Random Module
- Case Study: Calculator Example

Assignment Operator

- As mentioned in previous module

  The **=** sign is the assignment operator but not the equality in mathematics

- So, when we see

```
a = 5
a = a + 7
```

First, we create a variable called **a** and assign a value of 5 to it. Second, we recall the value of **a**, add 7 to it, and assign the expression result to **a**

Assignment Operator

- Basic syntax:

  Left Hand Side (LHS) = Right Hand Side (RHS)

- RHS is an ***expression*** and LHS is a ***variable***
- What "assignment" means is:

  1) Evaluate the expression on RHS

  2) Take the resulting value and associate (assign) it with the name (variable) on the LHS (in the namespace)

Examples

- Example: **`x = 2 + 3 * 5`**

  evaluate expression (2+3*5): 17

  update the value of **x** to reference 17

- Example (y has value 2): **`y = y + 3`**

  evaluate expression (y+3): 5

  update the value of **y** to reference 5

NOTE: If the variable name appears for the first time, create the variable in namespace!

Examples

- Examples:

`myInt = 5`	Ok
`myInt + 5 = 7`	Invalid Syntax
`myInt = print("hello")`	Invalid Syntax
`print myInt = 5`	Invalid Syntax

Why?

More Examples

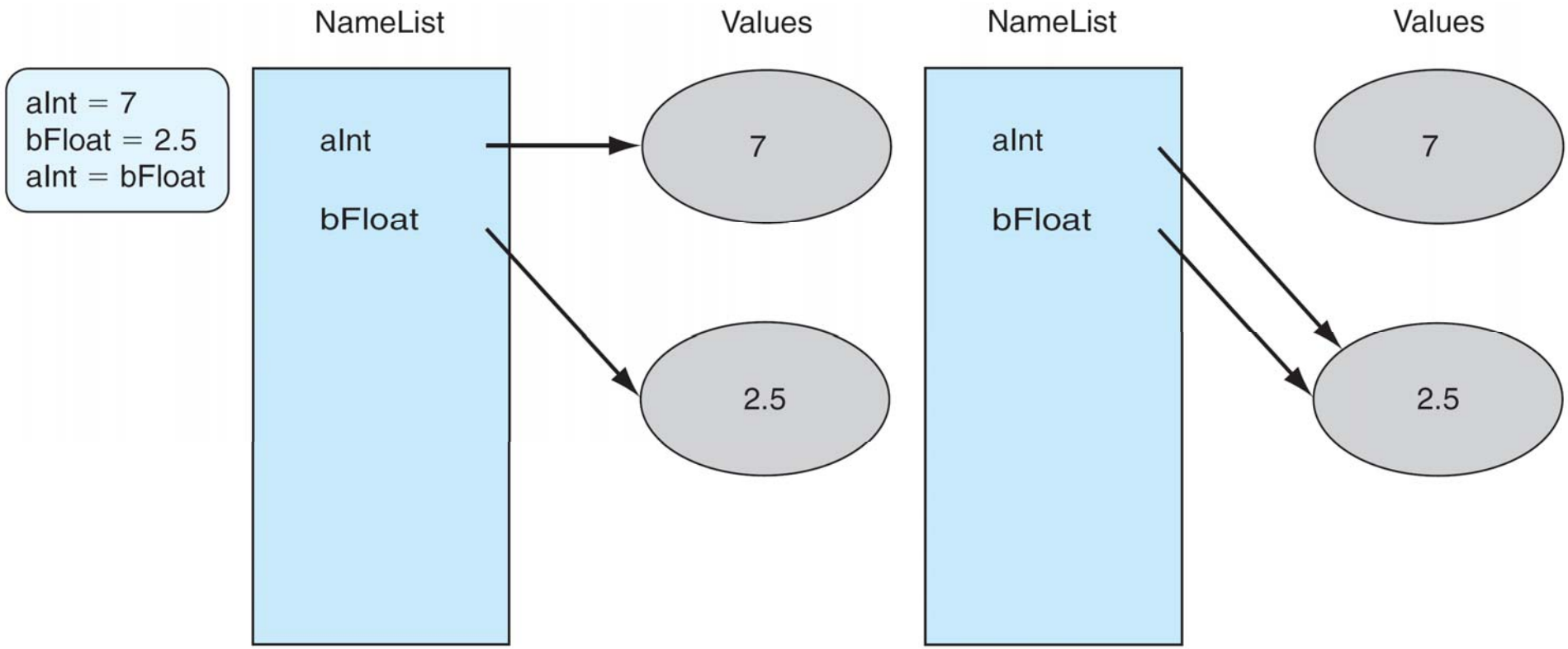


FIGURE 1.2 Namespace before and after the final assignment.

Both **`aInt`** and **`bFloat`** got the same reference!!!

Exercise: What printed out?

```
a = 2
b = 5

a = b    # same reference for both a and b
print( a )
print( b )

b = 7
print( a )
print( b )
```

Result… Reference is Tricky!!!

```
a = 2
b = 5

a = b    # same reference for both
print( a )
print( b )

b = 7
print( a )
print( b )
```

Result:

```
>>> a = 2
>>> b = 5
>>> a = b
>>> print( a )
5
>>> print( b )
5
>>> b = 7
>>> print( a )
5
>>> print( b )
7
```

A new reference

Chained assignment

- Let's say we want more variables to take the same value:

```
a = b = 1
a = b = c = 10
```

1. Don't make it too long or clumsy…
   Readability!!!
2. How about this? `a = b = 5 = c = 10`

Swapping two values

- Let's say we want to swap values

```
a = 1
b = 2
```

- Can we do this?

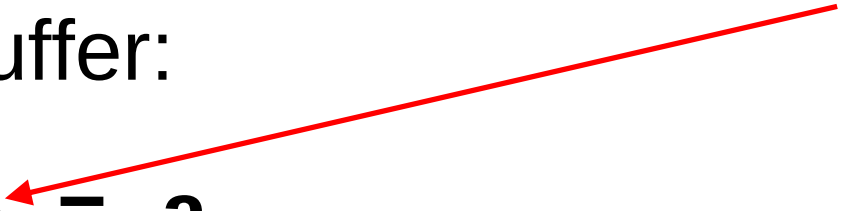
```
a = b
b = a
```

Then??? Correct???

Computationally incorrect!!! Why?

Swapping two values

- One standard way in many programming languages is to use a temporary variable as a buffer:



```
tmp = a
a   = b
b   = tmp
```

We can then swap the reference (computationally)!

Multiple Assignment

- But Python has a special feature

```
a = 1
b = 2
a,b = b,a
```

  - Make sure *same* number of elements on LHS and RHS
  - Python makes the tmp buffer for u (implicitly, hidden from your sight)

- Use comma for multiple assignment
- Swapping can be done in one line!!!
- Note: it supports more than two elements

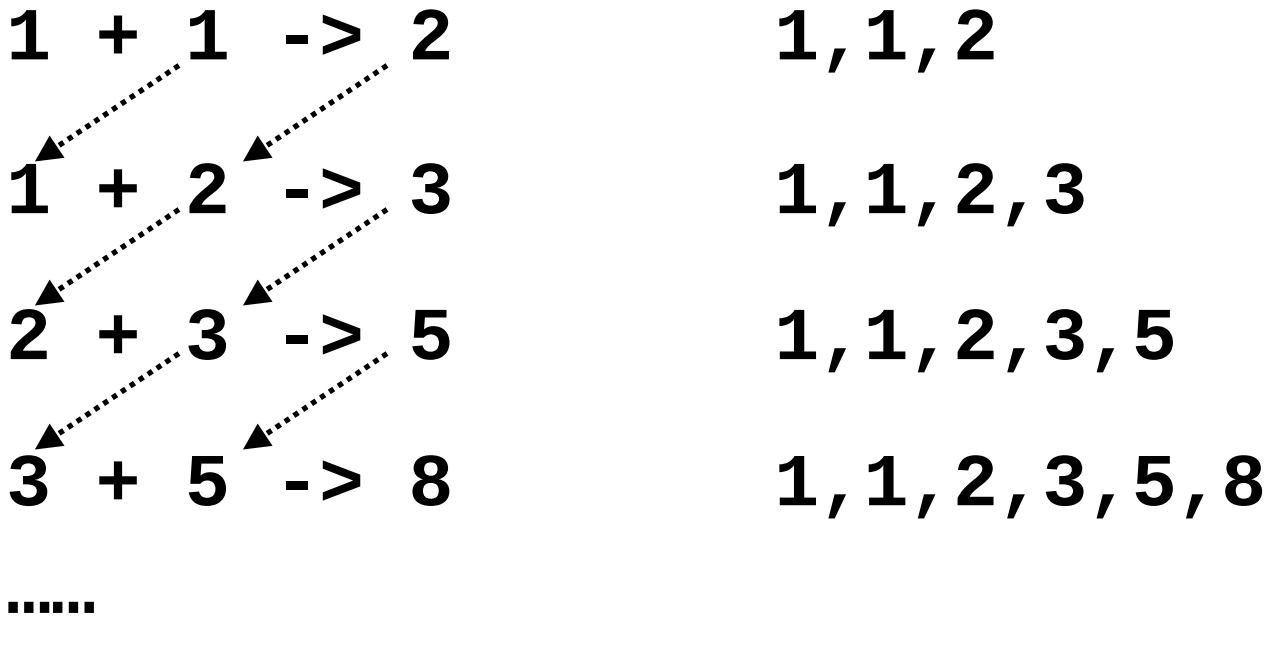
```
a,b,c = 10,11,12
```

CASE STUDY: Fibonacci sequence

- Problem: Generate the Fibonacci sequence

  1,1,2,3,5,8,13,21,…

- Mechanism



CASE STUDY: Fibonacci sequence

- Python implementation:
  - Use **a** and **b** to keep track of two consecutive values in the sequence for each iteration
  - and update them iteratively using multiple assignment

```
a = 1
b = 1
while a<1000:
    print( a )
    a,b = b,a+b
```

Python is simple and good for rapid prototyping

Topics

- Variables
- Assignment Operator
- **Data Types**
- Data Conversion
- Operators
- Powerful Data Types and Random Module
- Case Study: Calculator Example

Data types

- In Python, every “thing” is an object with type and name(s) (in case referenced by more than one variables), e.g.,
  - integers: 5
  - floats: 1.2
  - booleans: **`True`**
  - strings: “anything” or ‘something’
  - lists: [,]: [‘a’,1,1.3]
  - others we will see

In Python, both single & double quotes are for string

What is a Data Type?

- A type in Python essentially defines:
  - the internal structure (the kind of data it contains)
  - the kinds of operations you can perform
- **`'abc'.capitalize()`** is a method you can call on strings, but not integers
- Some types have multiple elements (collections), we'll see those later

Basic Types in Python

- Numbers:
  - Integers

    1, -27
  - Floating point numbers (Real)

    3.14, 10., .001, 3.14e-10, 0e0
  - Complex Numbers: 2 + 3j
- Booleans: **`True`**, **`False`**
- String and other types

Numbers: Integers

- Designated "**`int`**"
- Note: Python 3
  - *Unlimited* precision!!!!!!!

    1 + 1000 + 1000000000000000000000000
- Note: Python 2 (if you know…)
  - There are two categories of integers: "long" and "int", but they are combined into one type, basically "long," in Python 3

Numbers: Floating Point

- Designated "**`float`**"
- Floating point numbers are real numbers with decimal points:

  3.14, 10., .001, 3.14e-10, 0e0

- Values stored in floating point are usually approximated, e.g.,

```
>>> a = 2.0 / 3.0
>>> print(a)
0.6666666666666666
```

Integers have exact precision!!!
But not float…

Numbers: Complex Numbers

- Designated "**`complex`**"
- Python provides also complex numbers:

  real part + imaginary part J

  (J can be upper or lower case)

No space here

- Real and imaginary parts input can be in integer or floating point numbers
- No space before J

```
>>> 2 + 3j
(2+3j)
>>> 2.1 + 3J
(2.1+3j)
>>> 2 + 3.1j
(2+3.1j)
>>> 2 + 3 j
SyntaxError: invalid syntax
>>>
```

Numbers: Complex Numbers

- Use **`z.real`** and **`z.imag`** to extract the real and imaginary parts

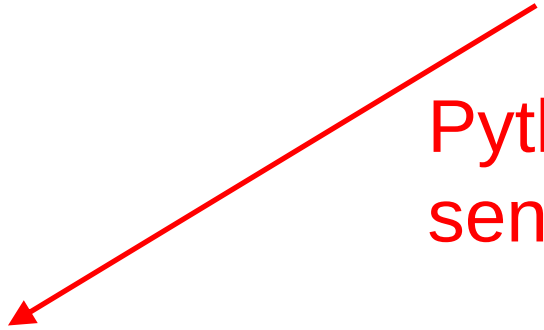
Note: we use the **dot operator** here to get the parameters (similar **syntax** like **`math.pi`** for modules)

```
>>> z = 2 + 3.1j
>>> z.real
2.0
>>> z.imag
3.1
>>>
```

Boolean

- Designated "**`bool`**"
- For logical operations, see Module 6
- Either **`True`** or **`False`** (capitalize T and F!)

```
>>> a = True
>>> print(a)
True
>>> b = False
>>> c = true
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    c = true
NameError: name 'true' is not defined
>>>
```



Python is case sensitive!!!

String

- Designated "**`str`**"
- First *collection* type you learnt
- Collection type contains *multiple* objects organized as a single object type
- String is basically a sequence, typically a sequence of characters delimited by single ('…') or double quotes ("…")

Duck-typing

- Compared to C and Java, you may wonder "How Python know the data types?"
- Python uses Duck-typing:

  "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

```
>>> a = 99
>>> b = 99.9
>>> c = '100'
>>> d = True
```

Four variables!
Their types?

Keep Track of Data Types!!!

- Python does not have variable declaration like Java or C to announce or create a variable
- We create a variable by just assigning a value to it and the type of the value defines the type of the variable
- If we re-assign another value to the variable, **its type can change**!!!
- So… KEEP TRACK!!!

Keep Track of Data Types!!!

- A variable in Python can have different type at different time

```
>>> myVar = 7
>>> myVar
7
>>> myVar = 7.77
>>> myVar
7.77
>>> myVar = True
>>> myVar
True
>>> myVar = "True"
>>> myVar
'True'
```

The type function

- In Python, the type function allows you to know the type of a variable or literal:

```
>>> type("abc")
<class 'str'>
>>> type("hello class")
<class 'str'>
>>> a = 100
>>> type(a)
<class 'int'>
>>> b = 1.2
>>> type(b)
<class 'float'>
```

```
>>> c = 1+2j
>>> type(c)
<class 'complex'>
>>> d = True
>>> type(d)
<class 'bool'>
>>> s = "wake up"
>>> type(s)
<class 'str'>
```

Textbook: Hungarian notation

- Variable naming: append the name of the type to the variable name:

```
>>> myVar_int   = 7
>>> myVar_float = 7.77
>>> myVar_bool  = True
>>> myVar_str   = "True"
```

Note the name!

```
import math
radiusString  = input("Enter the radius of your circle:")
radiusFloat = float(radiusString)
circumference = 2 * math.pi * radiusFloat
area = math.pi * radiusFloat * radiusFloat
```

Topics

- Variables
- Assignment Operator
- Data Types
- **Data Conversion**
- Operators
- Powerful Data Types and Random Module
- Case Study: Calculator Example

Data Conversion

- Converting a value to other type by *returning* a new value of that type
- Must be compatible and reasonable

  e.g., cannot convert **`"abc"`** to integer
- There are conversion operations associated with the type itself:

  `int(someVar)` converts and returns an integer

  `float(someVar)` converts and returns a float

  `str(someVar)` converts and returns a string

```
>>> myVar = '123'
>>> a = int(myVar)
>>> a
123
>>> b = float(a)
>>> b
123.0
>>> c = float(myVar)
>>> c
123.0
>>> s = str(9.99)
>>> s
'9.99'
>>> float(s)
9.99
>>> float("abc")
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    float("abc")
ValueError: could not convert string to float: 'abc'
```

What are the types?
Let's keep track of them!

```
>>> a = 1.1
>>> i = int(a)
>>> i
1
>>> b = 1.6
>>> i = int(b)
>>> i
1
>>> c = 1.99999999
>>> i = int(c)
>>> i
1
>>> d = -1.999999999
>>> i = int(d)
>>> i
-1
```

What happens if we convert a float to an integer?

Truncated!!!!!!!! (remove anything after the decimal point)

Want to round off? … **`int(a+0.5)`**

Topics

- Variables
- Assignment Operator
- Data Types
- Data Conversion
- **Operators**
- Powerful Data Types and Random Module
- Case Study: Calculator Example

What are Operators?

- For each type, there is a set of operators allowing us to *perform computation* on that type
- Some symbols are used for different purposes for different types; we call this operator ***overloaded*** (this is called ***overloading***)
  - E.g., “+” is addition for integers but concatenation for strings (see module 8)

Types of Operators

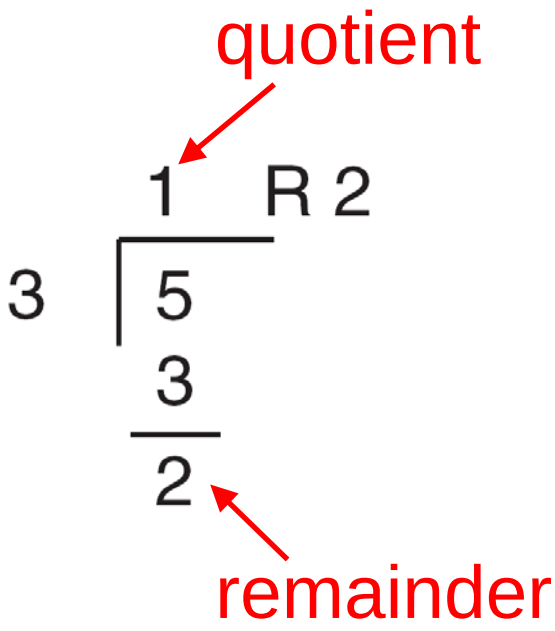
Basic types of operators:

1. Assignment operator = (covered already)
2. Arithmetic operators
3. Relational operators (comparison)
4. Logical operators
5. is operator

Note: bitwise operators – see next module

#2: Arithmetic Operators

- Integer operators
  - addition and subtraction: +, -
  - multiplication: *
  - division
    - quotient: /
    - remainder: %
  - exponent (exp): **
- Floating point operators
  - add, subtract, multiply, divide, exp: +, -, *, /, **



#2: Arithmetic Operators

Command	Name	Example	Output
+	Addition	3+4	7
-	Subtraction	9 -5	4
*	Multiplication	3*4	12
/	Division*	16/3	5
%	Remainder	16%3	1
**	Exponent	2**3	8

* Note: do not mix up / and \ (backslash)

Issue #1: Binary Operators

- *Binary* operators generally take two values (operands) of the **same** type * and return values of the **same** type

But… for division:

```
>>> a = 8.0 / 3.0
>>> print(a)
2.6666666666666665
>>> a = 8 / 3
>>> print(a)
2.6666666666666665
>>> type(a)
<class 'float'>
```

Integer division produces float!!! A new feature in Python 3 but not in Python 2, C/C++, etc.

Note: *Unary* operator – one value, e.g., -3 or +2.0

Issue #2: Division in Python

- Difference between Python versions:
  - In Python 2.x:

    Integer / integer → integer

    Integer division

    8 / 3 → 2
  - In Python 3.x:

    Integer / integer → float

    8 / 3 → 2.6666666666666665
- Note: Division by Zero
  - 1 / 0 -> run time error

A Trick in Python 3:

```
>>> 8/3
2.6666666666666665
>>> 8//3
2
```

Issue #3: Mixed Operations

- We have seen 8/3 and 8.0/3.0
  - How if 8 / 3.0? Different types: int and float
  - This is called mixed operation
- In Python
  - Python will **automatically convert** the data to the most *detailed* result. Thus, 8 → 8.0, and the result is 2.6666666
    - Detail: **`int < float`**
  - So… actually no mixed operations. Rather, data are **<u>implicitly</u>** converted.

Issue #4: Order of Calculation

- General mathematical rules apply:

Operators	Description
()	Parenthesis (grouping)
**	Exponentiation
+x, -x	Positive, Negative
*, /, %	Multiplication, division, remainder
+, -	Addition, Subtraction

- Note: always use parenthesis if in doubt… safe!!!

```
3 * (4 + 5) ** 2
```

Issue #5: Augmented Assignments

- These operations are *shortcut*
- Make the code easier to read

Shortcut		Equivalence
`myInt += 2`	↔	`myInt = myInt + 2`
`myInt -= 2`	↔	`myInt = myInt - 2`
`myInt /= 2`	↔	`myInt = myInt / 2`
`myInt *= 2`	↔	`myInt = myInt * 2`
`myInt %= 2`	↔	`myInt = myInt % 2`

#3: Relational Operators

- Compare two numbers (**`float`** or **`int`**) and return a **boolean**: either **`True`** or **`False`**

operator	example	meaning
==	a == 1	equal to
!=	b != 2	not equal to
<	c < 3	less than
<=	d <= 4	less than or equal to
>	f > 5.0	greater than
>=	f >= 6.0	greater than or equal to

#4: Logical Operators

- Logical operators *connect* boolean values and expressions and *return* a boolean value as a result: **`not`**, **`and`**, **`or`**

operator	example	meaning
not	not num < 0	Flip T/F
and	(num1 > num2) and (num2 >num3)	Return True only if both are True
or	(num1 > num2) or (num2 >num3)	Return True if either one is True

Examples

- Examples:

```
have_holidays == True and saving >= 10000

temperature > 37.5 and hasMedicine == False

MathScore < 50 or EngScore < 50 or …

MathScore < 50 and EngScore < 50 and …

Num % 2 == 0 and Num % 3 == 0
 -> Num % 6 == 0
```

What do they meaning?

Short circuit

Given **`p=True`**, **`q=True`**, **`r=False`**, **`s=False`**

- Short-circuit for **`and`**

  If we evaluate:

  `( p and q and r and s )`

  We know the expression is False when we reach r

- Short-circuit for **`or`**

  If we evaluate:

  `( s or r or q or p )`

  We know the expression is True when we reach q

Think about the logical meaning!!

http://docs.python.org/library/stdtypes.html#boolean-operations-and-or-not

Chained Comparisons

- In Python (not in most languages), chained comparisons work just like you would expect in a mathematical expression:
- Say, **`myInt`** has a value **`5`**

```
0 <= myInt and myInt <= 5
    True
0 <= myInt <= 5
    True
0 <= myInt <= 5 > 10
    False
```

same meaning (implicit "**and**")

Just apply each operator to compare its two neighboring values Even for this

Exercise

```
p = True
q = True

print( (not p) or q, (p and q) or q, (p or q) and p, (p or q) and (p and q) )

p = True
q = False

print( (not p) or q, (p and q) or q, (p or q) and p, (p or q) and (p and q) )

p = False
q = True

print( (not p) or q, (p and q) or q, (p or q) and p, (p or q) and (p and q) )

p = False
q = False

print( (not p) or q, (p and q) or q, (p or q) and p, (p or q) and (p and q) )
```

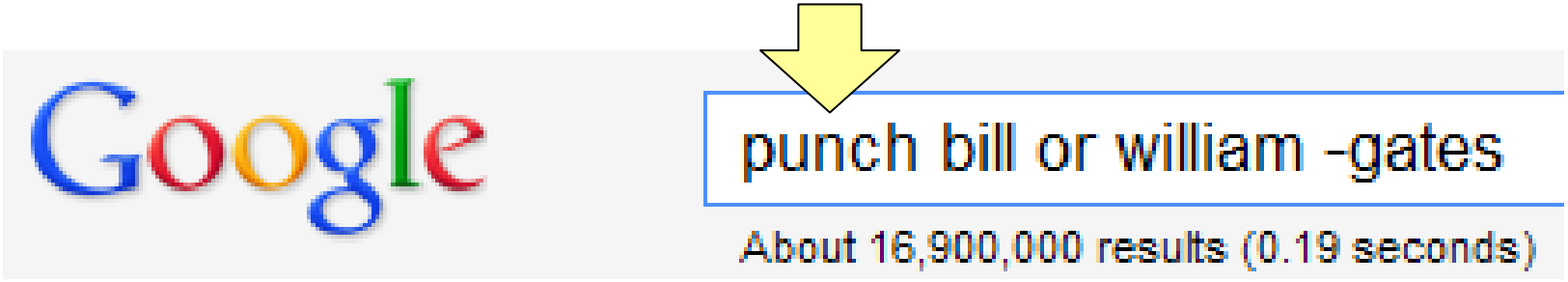
Want to check your answers? Just try it in Python

Cast study:
Google Search uses Booleans

- All terms are and'ed together by default
- You can specify or (using OR)
- You can specify not (using -)

Example is:

'Punch' and ('Bill' or 'William') and not 'gates'



#5: is operator

- Recall the namespace concept:

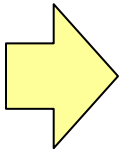
```
float1 = 2.5
float2 = 2.5
float3 = float2
```

- When we run "**`float3 = float2`**,"
  both **`float3`** and **`float2`** get the same reference

#5: is operator

- It checks if two variables have the same reference while == compares values only

```
>>> float1 = 2.5
>>> float2 = 2.5
>>> float3 = float2
>>> float1 == float2
True
>>> float3 == float2
True
>>> float1 is float2
False
>>> float3 is float2
True
```



```
>>> id(float1)
11392272
>>> id(float2)
16356152
>>> id(float3)
16356152
```

Function **`id`** returns the identity number of a variable

Topics

- Variables
- Assignment Operator
- Data Types
- Data Conversion
- Operators
- **Powerful Data Types and Random Module**
- Case Study: Calculator Example

Powerful Data types

- Some very powerful data types in Python
  - **List** – sequence of values
  - **Dictionary** – values with keys
  - **Set** – a collection of unique elements
  - **Point** – XYZ
  - **Line Segment** – two points

  (See textbook and Python website for detail; note: they are very useful; not examinable unless covered in 2nd part of this course)

A glimpse: List

- Python allows us to create a list of elements
  - Like string, a list is also a sequence but uses [ ]

```
>>> s1 = [ 1 , 6 , 2 , 3 ]
>>> s2 = [ True , False , True ]
>>> s3 = [ "apple" , "orange" , "banana" ]
>>> print( s1 )
[1, 6, 2, 3]
>>> print( s2 )
[True, False, True]
>>> print( s3 )
['apple', 'orange', 'banana']
```

Module random

- Provides some powerful functions:
  - **`randint( a , b )`**:

    Return a random integer in **`[a,b]`** (*inclusive*)
  - **`choice( x )`**:

    Return a random element in sequence **`x;`**
    `x` has to be non-empty
  - **`shuffle( x )`**:

    Shuffle the sequence **`x`** in place; **`x`** can be a string and can be a set

  You need to **`import random`**!!!

```
>>> import random
>>> random.randint( 1 , 100 )
25
>>> print( s3 )
['apple', 'orange', 'banana']
>>> print( random.choice( s3 ) )
banana
>>> print( random.choice( s3 ) )
apple
>>> print( s1 )
[1, 6, 2, 3]
>>> random.shuffle( s1 )
>>> print( s1 )
[3, 1, 2, 6]
>>> print( s3 )
['apple', 'orange', 'banana']
>>> random.shuffle( s3 )
>>> print( s3 )
['orange', 'banana', 'apple']
```

Different elements picked at different time (same statement)

s1 is changed!

But hold on…

- Is random really random?
- Computation are **deterministic**. Computers cannot generate **true** random numbers.
- So what?

  Pseudo random!!!

  * *Pseudo random number generator*

    created by mathematical functions !!

  (Example? See the end of next lecture module)

Topics

- Variables
- Assignment Operator
- Data Types
- Data Conversion
- Operators
- Powerful Data Types and Random Module
- **Case Study: Calculator Example**

Calculator Example

- Before the example, let's learn an interesting and powerful Python function: **exec**
- It takes a string as an input and execute it like a Python statement in the command shell

```
>>> a = 3
>>> print(a)
3
>>> exec("a = a + 3")
>>> print(a)
6
```

Calculator Example

- Why powerful? Check this:

```
result = 0

while True:

    # Read a math expression
    input_str = input("Enter a math. expression: ")

    # Execute the expression
    exec( "result = " + input_str )

    # Print the result
    print( "result=" , result )
```

User input

Execute it

Note: + is the concatentation operator to connect strings

Repeat the indented block

Calculator Example

```
>>> ========================================== RESTART =======
>>>
Enter a math. expression: 3 + 4 * 5
result= 23
Enter a math. expression: result + 10
result= 33
Enter a math. expression: result ** 2
result= 1089
Enter a math. expression:
```

- We can read user inputs from command shell and **dynamically** execute them like “code”!!!
- Not many programming languages can do this

Note: Python - interpreter!!!

Take Home Messages

- Variables:
  - Namespace and name reference concept
- Assignment Operator: Single, Chained, Multiple
- Data Types:
  - Numbers (integers and float), Booleans, String (collection types)
  - Duck-typing, type function, data conversion
- Operators
  - Division, mixed operators, order of operations and parentheses, augmented assignments, relational and logical operators
- Additional: list, random module, and **exec**

Reading Assignment

- Textbook

  Chapter 1: Beginnings

  1.4 to 1.7

  (and also Chapter 0.8 about representing data)