CS100 Python
Introduction to Programming

Lecture 22. Basic Concepts

Fu Song

School of Information Science and Technology

ShanghaiTech University

Learning Objectives

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

Python Program Structure

- A Python program is constructed from code blocks
- Block is a piece of Python program statements that is executed as a unit, i.e., module, function, class, etc
- Interactive session, statements are executed as they are typed in, until the interpreter is terminated
- Script file (xx.py), the interpreter reads statements from the file and executes them until end-of-file (EOF) is encountered

Python Program Structure

- **Each statement usually occupies a single line ending with the newline character (Pythonic)**

print("Hello World 1") ←

Newline **'\n'** characters

- **Multiple statements per line separated by semicolon ';'**

print("Hello World 1"); … ; print("Hello World 2")

- **Pythonic: PEP 8 -- Style Guide for Python Code**

https://www.python.org/dev/peps/pep-0008

Python Program Structure

- **Implicit Line Continuation:** any statement containing '(', '[', or '{' is presumed to be incomplete until all matched

```
>>> a = [1, 2, 3,
         4, 5, 6,
         7, 8, 9]
```

Newline **'\n'** characters after ' , '

- **Explicit Line Continuation: i**n cases where implicit line continuation is not readily available or practicable, you can specify a backslash '\' character

```
>>> a = 1 + 2 + 3\
       + 4 + 5 + 6\
       + 7 + 8 + 9
```

Newline **'\n'** characters after ' \ '

Python Program Structure

- **Comments: the hash character (#) signifies a comment.**
  The interpreter will ignore everything from the hash character through the end of that line

  #This is a comment

- **But, a hash character inside a string literal is protected, and does not indicate a comment**

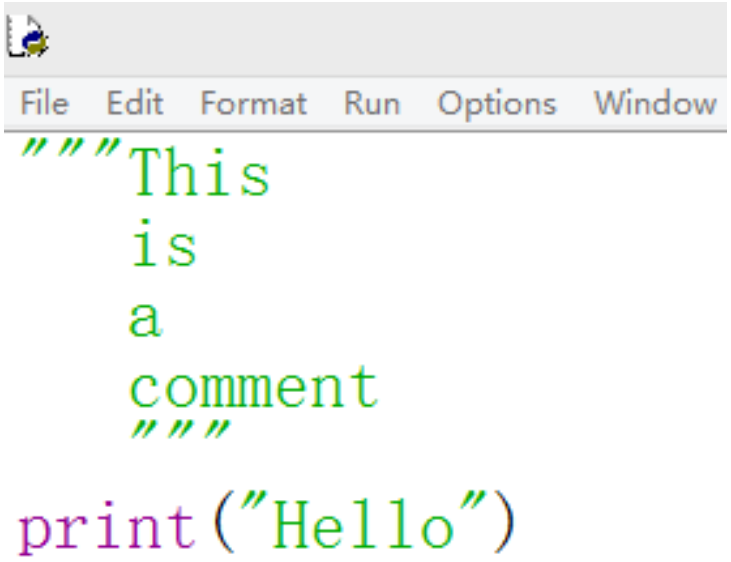
  "# This is not a comment, it is a string"

- **There is no multiline block comments like /* …*/ of C in Python**
- **Using multiple hash characters (#) for block comments**

Python Program Structure

- **Triple-quoted string:** ''' or """ can span multiple lines, it can effectively function as a multiline comment in script file ***.py, **not in interactive session**
- **But**, this is called **docstring** and used as a special comment at the beginning of a user-defined function that documents the function's behavior (to be **Pythonic**)

docstring will be explained later



Python Program Structure

- **Whitespace: almost always enhances readability in most programming languages including C/C++**

Character	ASCII Code	Literal Expression
space	32 (0x20)	' '
tab	9 (0x9)	'\t'
newline	10 (0xa)	'\n'

Python Program Structure

- **These programs are identical, whitespace are used for readability**

Python

```
>>> value1=100
>>> value2=200
>>> v=(value1>=0)and(value1<value2)
```

No whitespace there, otherwise syntax error

whitespace

Python

```
>>> value1 = 100
>>> value2 = 200
>>> v = (value1 >= 0) and (value1 < value2)
```

Python Program Structure

- **Whitespace as Indentation**
  - **There is one more important situation in which whitespace is significant in Python code Indentation**
  - **whitespace that appears to the left of the first token on a line—used to compute a line's indentation level, which in turn is used to determine grouping of statements (will see soon)**

Python

```
>>> print('foo')
foo
>>>     print('foo')

SyntaxError: unexpected indent
```

Syntax error

Learning Objectives

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

Everything in Python is an object

- **Objects** are Python's abstraction for data
- All data in a Python program is represented by **objects** or by **relations between objects**
- Every object has an **identity**, a **type** and a **value**
  - Constants
  - Statements
  - Functions
  - Instances of Classes
  - Classes
  - Modules
  - ……

Constants

Constants are objects of Built-in types

- **Boolean**: True, False
- **Integer**: 2,3,4, -1,-2
- **Float**: 3.14
- **Complex**: 1+2j, 2-4j

} unbounded

- **String**: ‘ab’, “aaa”
- **None**: a single value of NoneType
- **NotImplemented:** a single value of NotImplemented Type, return is value if the function is not implemented

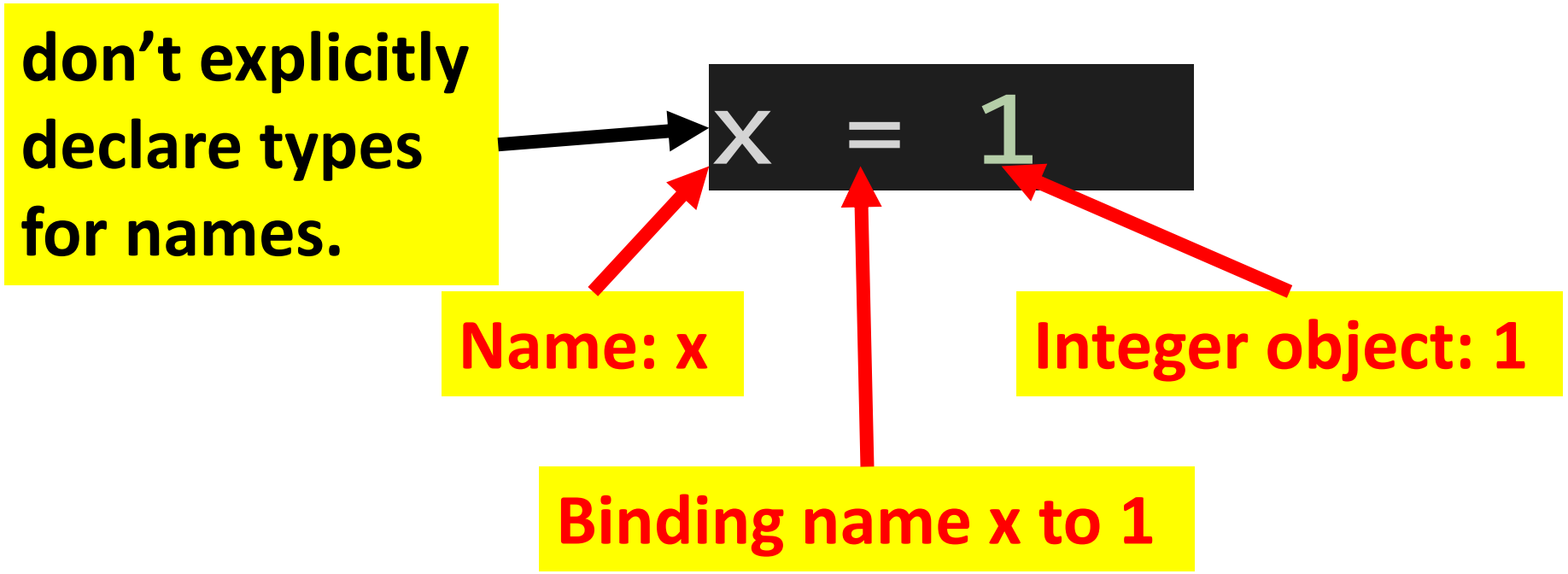
Constants

Types are inferred by the interpreter

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type(1+1.0j)
<class 'complex'>
>>> type('abc')
<class 'str'>
```

Names

- Names (i.e., variables, function) refer to objects
- Names are introduced by name binding operations
- Assignment binds a variable to an object
- There are many binding ways: parameter pass, return



Names

- The type of a name is inferred at runtime
- Determined by on the bounded object

Names

- The type of a name is inferred at runtime
- Determined by on the bounded object
- A name can bind to other objects later
- Hence, a name can have different types in the same program at different runtime

```
x = 1
x = 1.0
```

Binding name x to 1, x has type ‘int’

Binding name x to 1.0, x has type ‘float’

1 and 1.0 are different objects and types

Names: Example 1

```
x = 1
print("1: x has type:",type(x))
x = 1.0
print("1.0: x has type:",type(x))
x = 1 + 1.0j
print("1 + 1.0j: x has type:",type(x))
```

Output

```
1: x has type: <class 'int'>
1.0: x has type: <class 'float'>
1 + 1.0j: x has type: <class 'complex'>
```

Names

- **Get identity of an object via function**

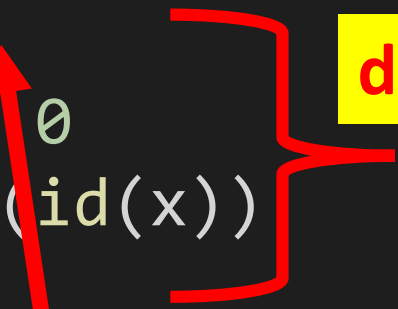
  id(object)

  - ✓ **Return the identity of an object**
  - ✓ **Identity is an integer which is guaranteed to be unique and constant for this object during its lifetime**
  - ✓ **Two objects with non-overlapping lifetimes may have the same identity**

Names: Example 2

```
>>> print(id(1))
1480145152
>>> x = 1
>>> print(id(x))
1480145152
>>> y = 1
>>> print(id(y))
1480145152
```

```
>>> print(id(1.0))
52630336
>>> x = 1.0
>>> print(id(x))
53366768
>>> y = x
>>> print(id(y))
53366768
```



x and y bind to the same object id: 1480145152

Lifetime of 1.0 terminates here, as no name binds to 1.0

Type Conversion

- a.k.a. type casting
- int to float/complex and float to complex

  ok
- float to int

  causes truncation
- complex to int/float

  impossible
- Str <=> int/float/complex

  ok

Type Conversion: Example 1

```
>>> x = 2
>>> float(x)
2.0
>>> y = 2.1
>>> int(x)
2
>>> z = 1 + 0j
>>> float(z)
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module> float(z)
TypeError: can't convert complex to float
>>> int(z)
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module> int(z)
TypeError: can't convert complex to int
```

ok

Truncation

Impossible

Explicit conversion

Type Conversion: Example 2

```
>>> str(1)
'1'
>>> str(1.0)
'1.0'
>>> str(1+0j)
'(1+0j)'
```

ok

Explicit conversion

In Python, there is implicit (i.e., assignment) type conversion as in C/C++

Expressions

- An expression is any combination of variables, constants and operators that can be evaluated to yield a result, similar to C/C++
- You can tell the interpreter explicitly how you want an expression to be evaluated by using parentheses (and)
- To make your code easier to read and maintain, you should be explicit and indicate with parentheses whenever possible

Boolean Expressions

- Boolean value: True, False
- Non-zero numeric values and NotImplemented are True
- Zero numeric values and None are False
- Boolean operations:

ascending priority 

- not has a lower priority than non-Boolean operators,

so (a=1, b=2)

1. not a == b is interpreted as not (a == b)
2. and a == not b is a syntax error

注意要加括号

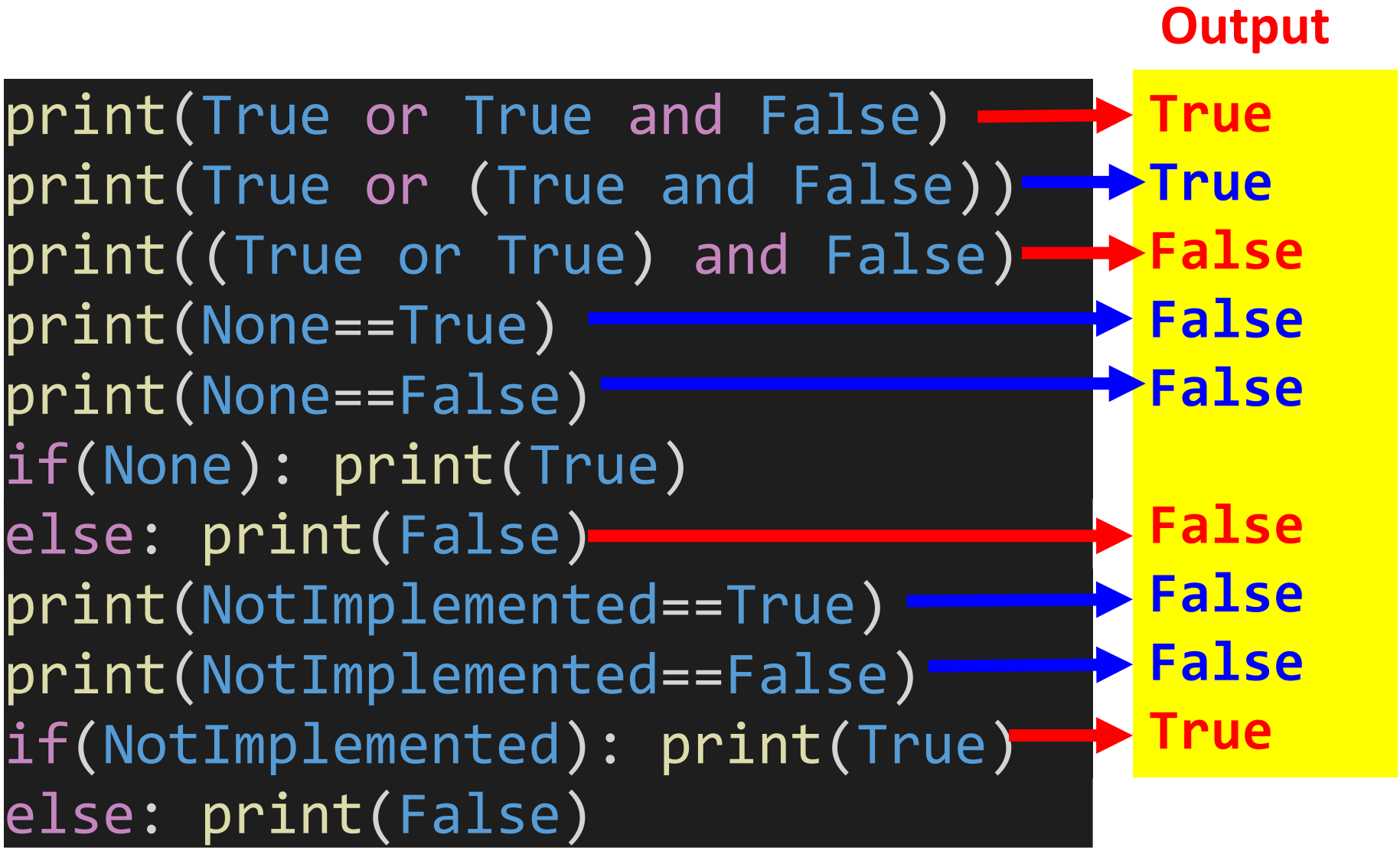
Boolean Expressions

- **e1 and e2 : it only evaluates e2 if e1 is true**
- **e1 or e2 : it only evaluates e2 if e1 is false**

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

Boolean: Example



Comparing None/NotImplemented with True/False results in False

Comparison operations

- They all have the same priority (which is higher than that of the Boolean operations)

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Comparison: Example

```
>>> 1 == 1.0
True
>>> 1 is 1.0
False
>>> 1 !=1.0
False
>>> 1 is not 1.0
True
```

Numeric operations

- They have a higher priority than comparison operations

ascending priority

Operations	Int	Float	Complex
+	add		
–	minus		
*	times		
/	division (result: Float)		division
//	Division (result: integer part)		–
%	remainder		–
abs()	Absolute		Magnitude
**	power		

Remark: float calculation is approximation: 1.2-1=0.19999

Bitwise Operations

- The priorities of the binary bitwise operations are all lower than numeric operations and higher than comparisons
- ~ has the same priority as the other unary numeric operations +, -

ascending priority

Operation	Result
x | y	bitwise or of x and y
x ^ y	bitwise exclusive or of x and y
x & y	bitwise and of x and y
x << n	x shifted left by n bits (n>0)
x >> n	x shifted right by n bits (n>0)
~x	the bits of x inverted

x << n ⇔ x * (2**n) x >> n ⇔ x // 2**n

Implicit Type Conversion

Arithmetic conversion

- in mix operation it converts the operands to be type of the higher ranking of the two
- int to float/complex and float to complex: following the same rules of explicit conversion

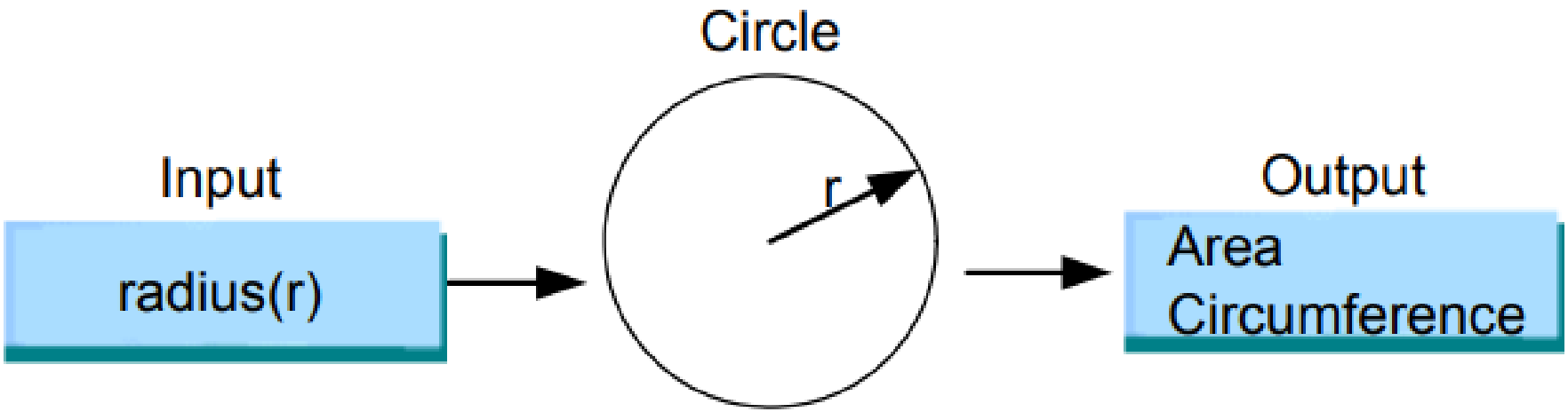
```
>>> 1 + 0.0
1.0
>>> 1 + (0+0j)
(1+0j)
>>>
```

int to float

int to complex

Remark: there is no assignment conversion

Example Circle



Area = π*r*r

Cirumference= 2*π*r

Example in C

```c
#include <stdio.h>
int main(){
    const float PI = 3.14;
    float radius, area, circumference;
    // Read the radius of the circle
    printf("Enter the radius: ");
    scanf("%f", &radius);
    // Calculate the area
    area = PI * radius * radius;
    // Calculate the circumference
    circumference = 2 * PI * radius;
    // Print the area and circumference of the circle
    printf("The area is: %0.10f\n", area);
    printf("The circumference is: %0.10f\n", circumference);
    return 0;
}
```

Example in Python

```
# circle.py
PI = 3.14
# Read the radius of the circle
radius = float(input("Enter the radius: "))
# Calculate the area
area = PI * radius * radius
# Calculate the circumference
circumference = 2 * PI * radius
# Print the area and circumference of the circle
print("The area is:", area)
print("The circumference is:", circumference)
```

Example Result

C

```
Enter the radius: 4
The area is: 50.2400016785
The circumference: is
25.1200008392
```

```
Enter the radius: 3
The area is: 28.2600002289
The circumference: is 18.840000
1526
```

Python

```
Enter the radius: 4
The area is: 50.24
The circumference is: 25.12
>>>
```

```
Enter the radius: 3
The area is: 28.259999999999998
The circumference is: 18.84
>>>
```

Warning

Floating-point arithmetic (FP) is arithmetic using formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision

https://en.wikipedia.org/wiki/Floating-point_arithmetic

Learning Objectives

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

Assignment

assignment_stmt ::=
target (",” target)* “=" expr (",” expr)*

names

expressions

- **An assignment statement 1) evaluates all the expressions and 2) assigns the results object to each of the target lists, from left to right**
- **Assignment is defined recursively depending on the form of the target (list)**

Augmented Assignment

assignment_stmt ::= target **augop** expr
augop::= **"+="** | **"-="** | **"*="** | **"/="** | **"//="**
|**"%="** | **"**="** | **">>="** | **"<<="**
| **"&="** | **"^="** | **"|="**

- **Same as in C/C++**
- target **x=** expr ⇔ target **=** target **x** expr

Assignment: Example

```
>>> x,y = 1, 2
>>> x
1
>>> y
2
>>> y,x = x,y
>>> x
2
>>> y
1
```

Swap values

Learning Objectives

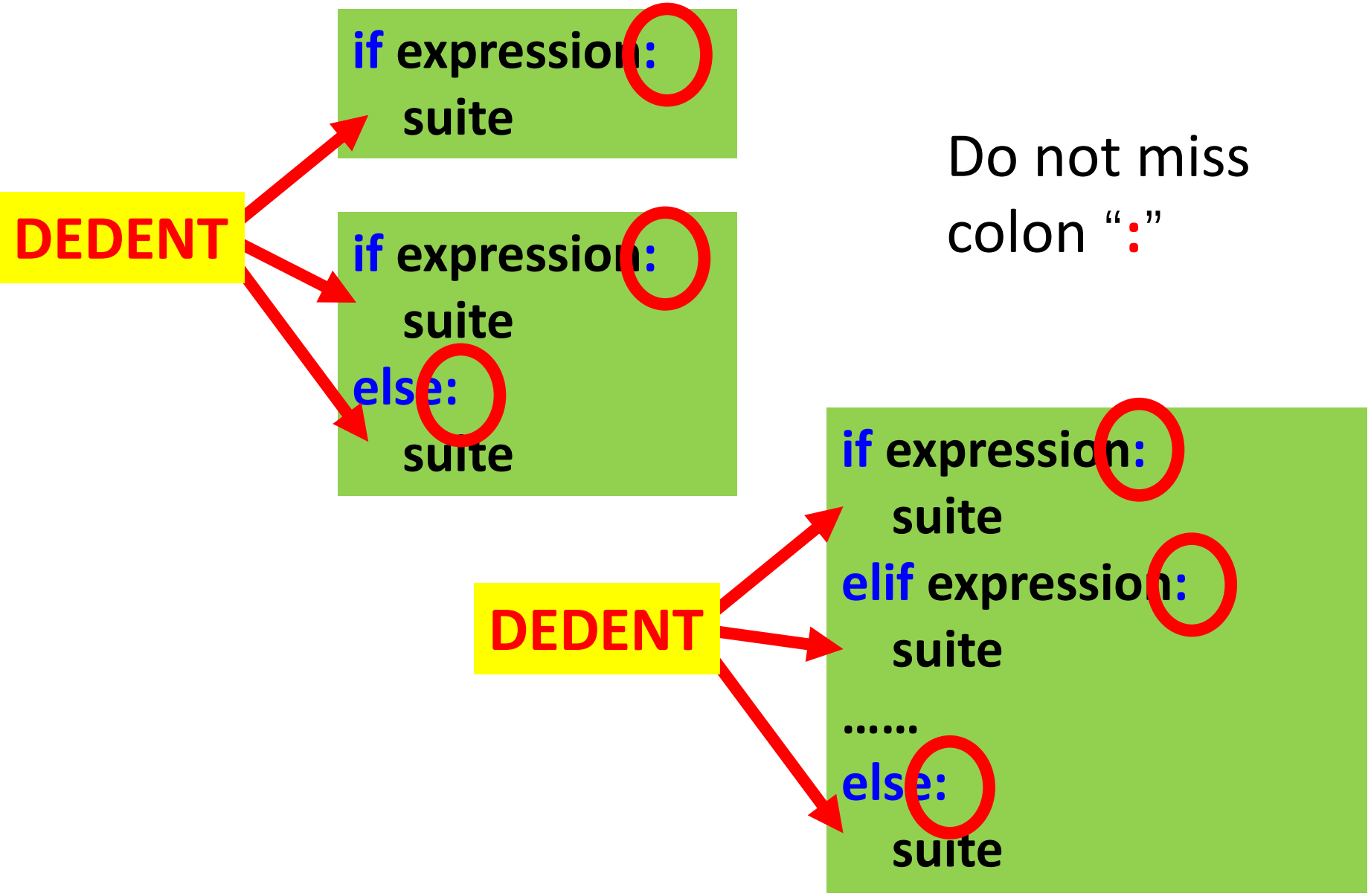
- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

If-else

```
if_stmt  ::=  "if" expression ":" suite
              ( "elif" expression ":" suite )*
              ["else" ":" suite]
suite    ::=  stmt_list "\n"  |  "\n" INDENT stmt_list + DEDENT
stmt_list  ::=  stmt (";" stmt)* [";"]
```

- It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true then that suite is executed (and no other part of the if statement is executed or evaluated)
- If all expressions are false, the suite of the else clause, if present, is executed
- **DEDENT:** 4 spaces per indentation level, **don't use tab**

If-else


if expression:
suite
DEDENT
if expression:
suite
else:
suite
Do not miss colon ":"
if expression:
suite
elif expression:
suite
.....
else:
suite
DEDENT

If-else: Example

```
score = float(input("Input your score:"))
if 90 <= score <= 100:
    print("A"); print("Well-done!")
elif 80 <= score < 90:
    print("B")
    print("Not too bad!")
elif 70 <= score < 80:
    print("C")
elif 60 <= score < 70:
    print("D")
else: print("Fail")
```

Same block use same spaces

DEDENT: 4 spaces per indentation level

If-else: Example

```
score = float(input("Input your score:"))
if 90 <= score <= 100:
print("A"); print("Well-done!")
elif 80 <= score < 90:
    print("B")
    print("Not too bad!")
elif 70 <= score < 80:
    print("C")
elif 60 <= score < 70:
    print("D")
else: print("Fail")
```

SyntaxError: expected an indented block

If-else: Example

```
score = float(input("Input your
score:"))
if 90 <= score <= 100:
    print("A"); print("Well-done!")
elif 80 <= score < 90:
  print("B")
    print("Not too bad!")
elif 70 <= score < 80:
    print("C")
elif 60 <= score < 70:
    print("D")
else: print("Fail")
```

SyntaxError: Unexpected Indent

If-else: Example

```
score = float(input("Input your score:"))
if 90 <= score <= 100:
    print("A"); print("Well-done!")
elif 80 <= score < 90:
    print("B")
    print("Not too bad!")
elif 70 <= score < 80:
  print("C")
elif 60 <= score < 70:
 print("D")
else: print("Fail")
```

Different blocks can have distinct Indent

Pythonic: Use same number of spaces at same level

Learning Objectives

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

For Loop

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

- The expression list is evaluated once and yield an iterable object
- Each item in turn is assigned to the target list using the standard rules for assignments and then the suite is executed
- The suite is then executed once for each item provided by the iterator, in the order returned by the iterator
- When the sequence is empty or an iterator raises a StopIteration exception), the suite in the else clause, if present, is executed, and the loop terminates

For Loop: Example

```
for i in [0,1,2,3,4]:
    print(i)
```

```
for i in range(5):
    print(i)
```

i is an iterator object of integers from 0 up to 4

Output

```
0
1
2
3
4
>>>
```

For Loop: Example 2

```
for i in range(5):
    print(i)
else:
    print("stop after:",i)
```

Output

```
0
1
2
3
4
stop after: 4
>>>
```

Range

Arguments of range must be integers

range(stop) -> range object

- **Iterators of integers from 0 to stop-1**

range(start, stop[, step]) -> range object

- **Iterators of integers from start to stop-1 with step step**
- **In default, step is 1**

  start, start+step, start+2*step,…., start+n*step<stop

For Loop: Example 2

```
for i in range(2,10,3):
    print(i)
else:
    print("stop after:",i)
```

Output

```
2
5
8
stop after: 8
>>>
```

Break and Continue

To alter flow of control inside loop

- **Execution of break causes immediate termination of the inner most enclosing loop**
- **Execution of continue causes all subsequent statements after the continue statement are not executed for this particular iteration**

For Loop: Example 3

```
for i in range(5):
    if(i==3):
        break
    print(i)
else:
    print("stop after:",i)
```

Output

```
0
1
2
>>>
```

"else" is **not** executed after **break**

For Loop: Example 4

```
for i in range(5):
    if(i==3):
        continue
    print(i)
else:
    print("stop after:",i)
```

Output

```
0
1
2
4
stop after: 4
>>>
```

"else" is executed after **continue**

While Loop

```
while_stmt ::=  "while" expression ":" suite
                ["else" ":" suite]
```

This repeatedly tests the expression and,

- if it is true, executes the first suite;
- if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates
- Break and continue have same effects as in for loop

While Loop: Example 1

```
i = 0
while(i<5):
    print(i)
    i += 1
else:
    print("Stop at:",i)
```

Output

```
0
1
2
3
4
Stop at: 5
>>>
```

While Loop: Example 2

```
i = 0
while(i<5):
    print(i)
    i += 1
    if(i==3):
        break
else:
    print("Stop at:",i)
```

Output

```
0
1
2
```

"else" is **not** executed after **break**

While Loop: Example 3

```
i = 0
while(i<5):
    if(i==3):
        i += 1
        continue
    print(i)
    i += 1
else:
    print("Stop at:",i)
```

Output

```
0
1
2
4
Stop at: 5
>>>
```

"else" is executed after **continue**

While Loop vs. For Loop

- For loop knows the number of times of the loop
  - ✓based on a generator or a sequence of items
  - ✓always terminate
- While loop does not know the number of times
  - ✓based on a condition (True or False)
  - ✓may not terminate (infinite)
- Any for loop can be converts into while loop
- It is better to use for loop when it is possible

Learning Objectives

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements

The import statement

```
import_stmt     ::=
       "import" module ["as" identifier]
     | "from" relative_module "import" identifier ["as" identifier]

module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
```

1. Find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the import statement occurs
3. Multiple clauses (separated by commas), the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements

The import statement

```
import time
```

Import module time, which import all public names in time

```
start=time.process_time()
fib(40)
end=time.process_time()
print("Time cost",end-start,"s")
```

Access names in module via

modulename.name

The import statement

```
import time as TM

start=TM.process_time()
for i in range(100):
    pass
end=TM.process_time()
print("Time cost",end-start,"s")
```

Renamed the module time as TM

Access names in module via NewModuleName. name

The import statement

```
from time import process_time


start=process_time()
for i in range(100):
    pass
end=process_time()
print("Time cost",end-start,"s")
```

Import **process_time** from the module time

Can only access the imported name via importedname

The import statement

```
from time import process_time as GetTime

start=GetTime()
for i in range(100):
    pass
end=GetTime()
print("Time cost",end-start,"s")
```

Renamed process_time as GetTime

Can only access the imported name via NewName

The import statement

```
import_stmt     ::=
       "import" module ["as" identifier] ("," module ["as" identifier])*
     | "from" relative_module "import" identifier ["as" identifier]
                ("," identifier ["as" identifier])*
     | "from" module "import" "*“

module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
```

- One can import more than one modules in one import statements, or use ‘*’
- But, it is not recommended to do so

The assert statement

```
assert_stmt ::=  "assert" expression ["," expression]
```

assert expression ⇔ if __debug__:
    if not expression: raise AssertionError

- **assert** is a keyword
- **__debug__** is a built-in name which is **True** under normal circumstances, **False** when optimization is requested (command line option -O)
- **raise** is a keyword: raises an exception object following it
- **AssertionError:** is an exception object

The assert statement

```
def Div(x,y):
    assert y!=0
    return x/y

x = int(input("Input numerator:"))
y = int(input("Input denominator:"))
print(Div(x,y))
```

Input/Output

```
Input numerator:1
Input denominator:0
Traceback (most recent call last):
...
  File "C:\Users\Fu Song\Desktop\hello.py", line 2, in Div
    assert y!=0
AssertionError
```

The assert statement

assert_stmt ::= "assert" expression ["," expression]

```
assert expr1, expr2
```

⇔

```
if __debug__:
    if not expression: raise AssertionError(expr2)
```

Expr2: can be seen as an error message

The assert statement

```
def Div(x,y):
    assert y!=0, "denominator is 0"
    return x/y

x = int(input("Input numerator:"))
y = int(input("Input denominator:"))
print(Div(x,y))
```

Input/Output

```
Input numerator:1
Input denominator:0
Traceback (most recent call last):
...
  File "C:\Users\Fu Song\Desktop\hello.py", line 2, in Div
    assert y!=0, "denominator is 0"
AssertionError: denominator is 0
```

Recap

- Python Program Structure
- Understand and use
  - ✓ Object, name, expression
  - ✓ Control flow
    - Assignment
    - If-else
    - For/while loop
    - Break
    - Continue
    - Advanced statements