CSCI 141

Lecture 4:
More Variables
Operators and Operands
Code Execution: Statements and Expressions

Announcements

Announcements

One small syllabus change:

Announcements

One small syllabus change:

1. Previously: drop up to 9 missed poll questions. Now: poll questions are batched by day; drop up to 3 days of missed polls.

QOTD

What does the following code print?

```
print(int(3.91))
```

QOTD

Which of the following programs does not print the same thing as the others?

A:
```
a = 14
b = 3
print(a, b)
```

B:
```
a = 3
b = 14
print(14, 3)
```

C:
```
a = 14
b = a
print(a, b)
```

D:
```
a = 3
b = 14
print(14, a)
```

QOTD

Which of the following programs does not print the same thing as the others?

A:
```
a = 14
b = 3
print(a, b)
```
14, 3

B:
```
a = 3
b = 14
print(14, 3)
```

C:
```
a = 14
b = a
print(a, b)
```

D:
```
a = 3
b = 14
print(14, a)
```

QOTD

Which of the following programs does not print the same thing as the others?

A:

```
a = 14
b = 3
print(a, b)
```

14, 3

B:

```
a = 3
b = 14
print(14, 3)
```

14, 3

C:

```
a = 14
b = a
print(a, b)
```

D:

```
a = 3
b = 14
print(14, a)
```

QOTD

Which of the following programs does not print the same thing as the others?

A:
```
a = 14
b = 3
print(a, b)
```
14, 3

B:
```
a = 3
b = 14
print(14, 3)
```
14, 3

C:
```
a = 14
b = a
print(a, b)
```
14, 14

D:
```
a = 3
b = 14
print(14, a)
```

QOTD

Which of the following programs does not print the same thing as the others?

A:
```
a = 14
b = 3
print(a, b)
```
14, 3

B:
```
a = 3
b = 14
print(14, 3)
```
14, 3

C:
```
a = 14
b = a
print(a, b)
```
14, 14

D:
```
a = 3
b = 14
print(14, a)
```
14, 3

Goals

- Understand how to use variables in assignment statements and elsewhere in place of values
- Know the rules for naming variables, and the conventions for deciding on good variable names
- Know how to use the `sep`, and `end` keyword arguments with the `print` function.
- Know the definition and usage of operators and operands
  - Know how to use the following operators:
    `=, +, -, *, **, /, //, %`
- Understand the distinction between a statement and an expression.
- Understand function calls as expressions that evaluate to their return values.

Last time…

- A variable is a name in a program that refers to a piece of data (or a value).
- How do you use them?
  1. Decide what value you want to store in the variable
  2. Decide on a sensible name
  3. In your program, use the **assignment operator** to store that value in the variable:

```
my_age = 32
```

The assignment operator.

Last time: How to read an assignment statement

- Assigning a value is **not** stating an equality, like in math: it's storing a value.

```
my_age = 31
my_age = 32
```

A variable's value can be **updated** (overwritten) by a new value using the assignment operator.

"my_age gets 32"

"my_age equals 32"

"the variable my_age takes on the value 32"

"my_age becomes 32"

Last time: How to read an assignment statement

- Assigning a value is **not** stating an equality, like in math: it's storing a value.

```
my_age = 31
my_age = 32
```

A variable's value can be **updated** (overwritten) by a new value using the assignment operator.

“my_age equals 32”

“my_age gets 32”

“my_age becomes 32”

“the variable my_age takes on the value 32”

Last time: How to read an assignment statement

- Assigning a value is **not** stating an equality, like in math: it's storing a value.

```
my_age = 31
my_age = 32
```

A variable's value can be **updated** (overwritten) by a new value using the assignment operator.

"my_age equals 32"

"my_age becomes 32"

"my_age gets 32"

"the variable my_age takes on the value 32"

Last time: How to read an assignment statement

- Assigning a value is **not** stating an equality, like in math: it's storing a value.

```
my_age = 31
my_age = 32
```

A variable's value can be **updated** (overwritten) by a new value using the assignment operator.

"my_age equals 32"

"my_age gets 32"

"my_age becomes 32"

"the variable my_age takes on the value 32"

Last time: How to read an assignment statement

- Assigning a value is **not** stating an equality, like in math: it's storing a value.

```
my_age = 31
my_age = 32
```

A variable's value can be **updated** (overwritten) by a new value using the assignment operator.

"the variable my_age takes on the value 32"

"my_age becomes 32"

What can you do with variables?

Use them anywhere you'd use a value!

```
print(5)
```

```
a = 5
print(a)
```

These two programs both print 5.

Variable Names

Variable Names

- How do you use variables?
  1. Decide what value you want to store in the variable
  2. **Decide on a sensible name**
  3. In your program, use the assignment operator to store that value in the variable

Variable Names

- How do you use variables?
  1. Decide what value you want to store in the variable
  2. **Decide on a sensible name**
  3. In your program, use the assignment operator to store that value in the variable
- Great power, great responsibility: variables names can be almost anything!

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- **Valid** variable names:
  - start with a letter or an underscore (_)
  - can contain any letters and digits
  - are case-sensitive (name is not the same as Name)
  - are not the same as any Python language **keywords** (words that already mean something else):

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

```
True   2plus2   a_number   firstOfThreeValues
```

Variable Names

- Great power, great responsibility: variables names can be almost anything!
- **Valid** variable names:
  - start with a letter or an underscore (_)
  - can contain any letters and digits
  - are case-sensitive (name is not the same as Name)
  - are not the same as any Python language **keywords** (words that already mean something else):

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

```
True   2plus2   a_number   firstOfThreeValues
```

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- **Valid** variable names:
  - start with a letter or an underscore (_)
  - can contain any letters and digits
  - are case-sensitive (name is not the same as Name)
  - are not the same as any Python language **keywords** (words that already mean something else):

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

`a_number`   `firstOfThreeValues`

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- **Valid** variable names:
  - start with a letter or an underscore (_)
  - can contain any letters and digits
  - are case-sensitive (name is not the same as Name)
  - are not the same as any Python language **keywords** (words that already mean something else):

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

`True`   `2plus2`   `a_number`   `firstOfThreeValues`

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- **Valid** variable names:
  - start with a letter or an underscore (_)
  - can contain any letters and digits
  - are case-sensitive (name is not the same as Name)
  - are not the same as any Python language **keywords** (words that already mean something else):

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

`a_number` `firstOfThreeValues`

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long
  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores

```
current_time          a4          hair_color
      midterm_exam_grade_as_a_percent
```

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long
  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores

✓ `current_time` `a4` `hair_color`

`midterm_exam_grade_as_a_percent`

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long
  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores

✓`current_time`　　`a4`　　✓`hair_color`

`midterm_exam_grade_as_a_percent`

Variable Names

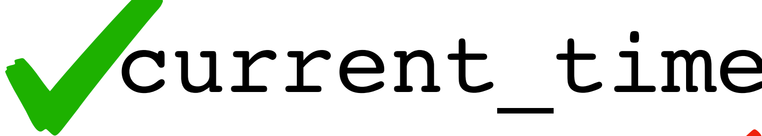
- Great power, great responsibility:
  variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long
  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores

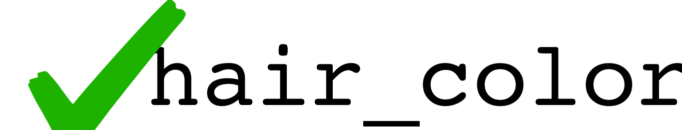
✓ `current_time` `a4` ✓ `hair_color`

✗ `midterm_exam_grade_as_a_percent`

Variable Names

- Great power, great responsibility: variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long
  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores





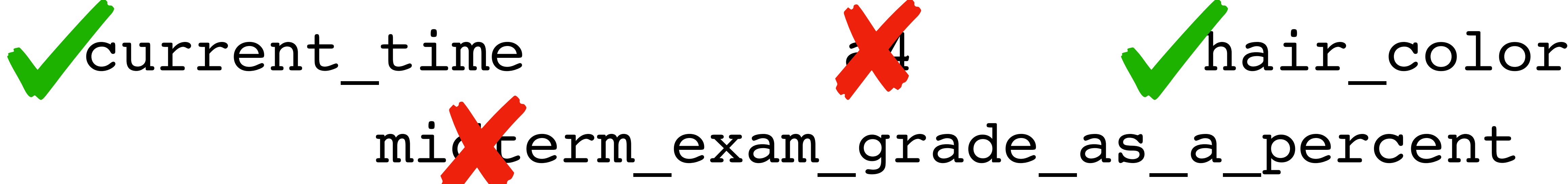
midterm_exam_grade_as_a_percent

Variable Names

- Great power, great responsibility:
  variables names can be almost anything!
- A **good** variable name:
  - is descriptive - tell a reader what data they refer to
  - is not too long

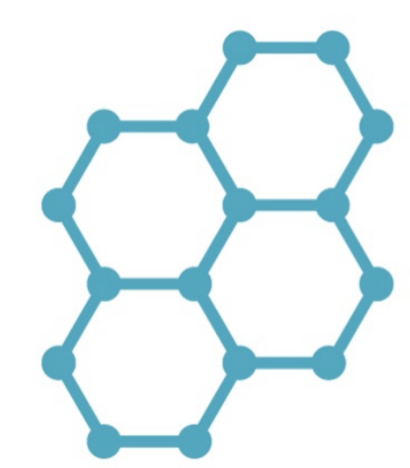
  these depend on context!

  - follows a standard naming convention, e.g.:
    - starts with lower case letter
    - words are separated by underscores



Variables and Assignment

What is the value of the variables a and b at the end of this program?



```
a = 5
b = 5
a = 6
b = 7
```

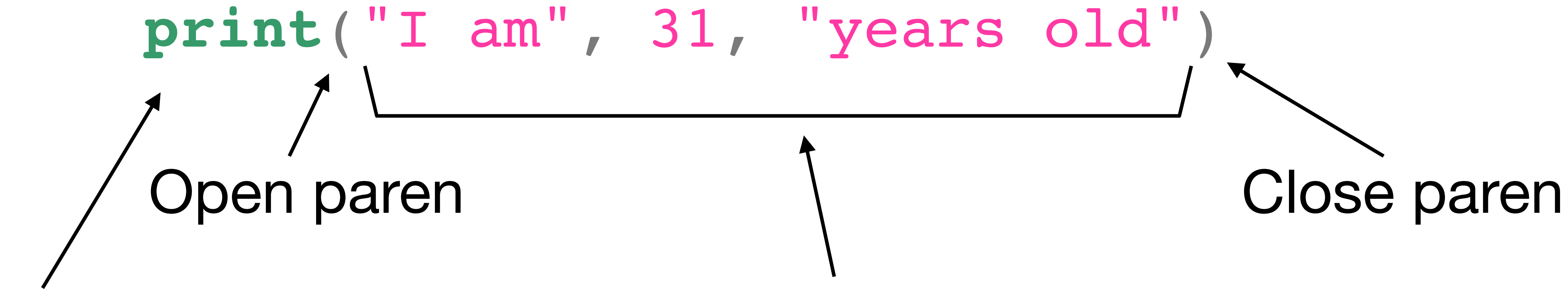
```
A. a: 5, b: 5

B. a: 5, b: 6

C. a: 7, b: 7

D. a: 6, b: 7
```

Aside: More on function calls...



Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", sep="_")
```

The `sep` keyword argument lets you specify what to print *between* arguments

Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", sep="_")
```

The `sep` keyword argument lets you specify what to print *between* arguments

Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", sep="_")
```

The `sep` keyword argument lets you specify what to print *between* arguments

If you leave it out, it's equivalent to passing a single space:

Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", sep="_")
```

The `sep` keyword argument lets you specify what to print *between* arguments

If you leave it out, it's equivalent to passing a single space:

```
print("Bellingham", "WA", "USA") # same as:
print("Bellingham", "WA", "USA", sep=" ")
```

Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", end="!")
```

The `end` keyword specifies what to print after the last argument.

Keyword Arguments

A mechanism for **optionally** passing information to a function.

```
print("Bellingham", "WA", "USA", end="!")
```

The `end` keyword specifies what to print after the last argument.

Demo: Print's Keyword Args

Demo: Print's Keyword Args

- Print with sep
- Print with end=""
  - End defaults to newline
- Print with end="!", end="!\n"
- Print with sep and end

The newline character

In a string, the special character sequence `\n` indicates a newline, or line break.

Example:

```
>>> print("line one\nline two")
line one
line two
>>>
```

Print's Keyword Args

Which of the following is printed by this line?

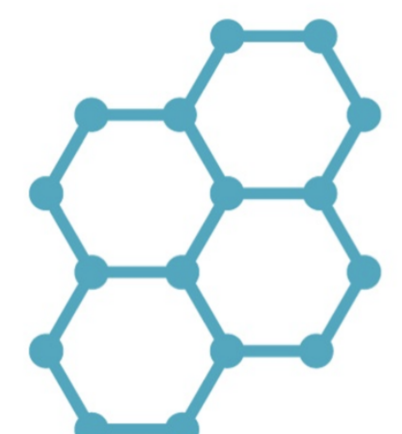
```
print("B", "C", "D", "BR", sep="A")
```

A. BACADABR

B. ABACADABRA

C. ABACADABR

D. BACADABRA



Print's Keyword Args

What is printed by the following code?

```
print("Name: ", end="\n---\n")
print("Date:", end="\n---\n")
```

A:

```
Name:
---

Date:
---
```

B:

```
Name:---Date:---
```

C:

```
Name:
---
Date:
---
```

D:

```
---
Name:
---
Date:
---
```

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

```
my_name = “Scott”
```
is an assignment statement

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

```
my_name = “Scott”
```
is an assignment statement

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

```
type(32)
2+2
int(a)
int(b) * 4
```
are all expressions

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

`my_name = “Scott”` is an assignment statement

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

```
type(32)
2+2
int(a)
int(b) * 4
```

are all expressions

The notation => is often used to mean “evaluates to”:

```
2 + 2 => 4
```

“two plus two evaluates to four”

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

`my_name = "Scott"` is an assignment statement

A statement in Python does not evaluate to a value!

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

```
type(32)
2+2
int(a)
int(b) * 4
```

are all expressions

The notation => is often used to mean "evaluates to":

```
2 + 2 => 4
```

"two plus two evaluates to four"

Statements and Expressions

- A statement is a line (or multiple lines) of code that Python can execute.

`my_name = “Scott”` is an assignment statement

A statement in Python does not evaluate to a value!

- An expression is a combination of values, variables, operators, and function calls that Python evaluates to determine its **value**.

```
type(32)
2+2
int(a)
int(b) * 4
```

are all expressions

The notation => is often used to mean “evaluates to”:

`2 + 2 => 4`

“two plus two evaluates to four”

Note: => is **not** a Python operator

Operators

- Operators are special symbols that represent computations we can perform.
- Operands are the values that an operator performs its computations on.
- We’ve seen one already: the assignment operator.

```
my_age = 32
```

Operators

- Operators are special symbols that represent computations we can perform.
- Operands are the values that an operator performs its computations on.
- We've seen one already: the assignment operator.

```
my_age = 32
```

The assignment operator.

Operators

- Operators are special symbols that represent computations we can perform.
- Operands are the values that an operator performs its computations on.
- We've seen one already: the assignment operator.

Its first (left) operand

```
my_age = 32
```

The assignment operator.

Operators

- Operators are special symbols that represent computations we can perform.
- Operands are the values that an operator performs its computations on.
- We've seen one already: the assignment operator.

Its first (left) operand

Its second (right) operand

```
my_age = 32
```

The assignment operator.

Operators

Some more Python operators:

```
=
+
-
*
/
**
//
%
```

Operators

Some more Python operators:

```
=
+
-
*
/
**
//
%
```

Some of these probably look familiar…

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

Operators

Some more Python operators:

= Assignment operator: stores a value in a variable

+ Addition
− Subtraction
* Multiplication

These ones do exactly what you think.

/ Division

**

//

%

Operators

Some more Python operators:

- = Assignment operator: stores a value in a variable
- + Addition
- - Subtraction
- * Multiplication
- / Division
- **
- //
- %

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

Operators

Some more Python operators:

`=` Assignment operator: stores a value in a variable

`+` Addition

`-` Subtraction

`*` Multiplication

`/` Division

`**`

`//`

`%`

This one too, with one quirk:
In Python, division **always** returns a `float`.

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

This one too, with one quirk:
In Python, division **always** returns a `float`.

```
3.0 / 2 => 1.5
```

Operators

Some more Python operators:

= Assignment operator: stores a value in a variable

+ Addition

- Subtraction

* Multiplication

/ Division

**

//

%

This one too, with one quirk:
In Python, division **always** returns a `float`.

```
3.0 / 2 => 1.5
7 / 2   => 3.5
```

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

This one too, with one quirk:
In Python, division **always** returns a `float`.

```
3.0 / 2 => 1.5
7 / 2   => 3.5
4 / 2   => ??
```

A. 2
B. 4
C. 2.0
D. 4.0

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**`
- `//`
- `%`

This one too, with one quirk:
In Python, division **always** returns a `float`.

```
3.0 / 2 => 1.5
7 / 2   => 3.5
4 / 2   => 2.0
```

A. 2
B. 4
C. 2.0
D. 4.0

Operators

Some more Python operators:

= Assignment operator: stores a value in a variable

+ Addition

- Subtraction

* Multiplication

/ Division

** Exponentiation

//

%

The exponentiation operator raises the left operand to the power of the right operand.

Operators

Some more Python operators:

- = Assignment operator: stores a value in a variable
- + Addition
- – Subtraction
- * Multiplication
- / Division
- ** Exponentiation
- //
- %

The exponentiation operator raises the left operand to the power of the right operand.

Math: $2^4 = 2 * 2 * 2 * 2 = 16$

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//`
- `%`

The exponentiation operator raises the left operand to the power of the right operand.

Math: $2^4 = 2 * 2 * 2 * 2 = 16$

Python: `2**4 => 16`

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//`
- `%`

The exponentiation operator raises the left operand to the power of the right operand.

Math: $2^4 = 2 * 2 * 2 * 2 = 16$

Python: `2**4 => 16`

Base Exponent

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

Integer division does division and evaluates to the integer **quotient**

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

Integer division does division and evaluates to the integer **quotient**

Math: 7 / 2 is 3 with remainder 1

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

Integer division does division and evaluates to the integer **quotient**

Math: 7 / 2 is 3 with remainder 1

Python: `7 // 2 => 3`

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

The modulus operator does division and evaluates to the integer **remainder**

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

The modulus operator does division and evaluates to the integer **remainder**

Math: 7 / 2 is 3 with remainder 1

Operators

Some more Python operators:

- `=` Assignment operator: stores a value in a variable
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Integer division
- `%` Modulus (remainder)

The modulus operator does division and evaluates to the integer **remainder**

Math: 7 / 2 is 3 with remainder 1

Python: `7 % 2 => 1`

Demo

- Arithmetic operators and expressions
  - `=, +, -, *, **, /, //, %`
- printing from a program vs evaluating expressions in the shell

Operator Practice

What does this expression evaluate to?

```
(9 % ( 5 // 1))
```

A: -1

B: 2

C: 4

D. None of the above

Operator Practice

What does this expression evaluate to?

```
(9 % ( 5 // 1))
```

A: -1

B: 2

C: 4

D. None of the above

Operator Practice

```
64 % 2

2**5

18 // 4

18 / 4

14 % 5
```

Operator Practice

```
64 % 2

2**5

18 // 4

18 / 4

14 % 5
```

Function Calls, Revisited

- Recall: function can take inputs called arguments
- New: A function can give back an output, called its return value.
- A function call is an expression that evaluates to the its return value.
  - `int(4.6)` evaluates to `4`
  - `print` does not return a value, so `print(4.6)` evaluates to `None`, a special keyword meaning no value

Fact

The `input` function's return value always has type `str`

Implication:

Fact

The `input` function's return value always has type `str`

Implication:

```
# ask for a number
a = input("Enter a number: ")
# but a is a string, so we need to:
user_number = float(a)
# now user_number has type float
```

Fact

The `input` function's return value always has type `str`

Implication:

```
# ask for a number
a = input("Enter a number: ")
# but a is a string, so we need to:
user_number = float(a)
# now user_number has type float

# we can do it in one line:
a = float(input("Enter a number:"))
```

Demo

Demo

- storing input's return value in a variable and converting its type
- function call with no return value
- expression on its own line in a program

Putting it all together

```
a = 4
b = float(2 + a)
```

Putting it all together

- Consider this program:

```
a = 4
b = float(2 + a)
```

- What happens when we execute it?

Putting it all together

- Consider this program:

```
a = 4
b = float(2 + a)
```

- What happens when we execute it?
  - the value 4 gets stored in a

Putting it all together

- Consider this program:

```
a = 4
b = float(2 + a)
```

- What happens when we execute it?
  - the value 4 gets stored in a
  - the expression 2+a is evaluated, resulting in the value 6

Putting it all together

- Consider this program:

```
a = 4
b = float(6)
```

- What happens when we execute it?
  - the value `4` gets stored in `a`
  - the expression `2+a` is evaluated, resulting in the value 6

Putting it all together

- Consider this program:

```
a = 4
b = float(6)
```

- What happens when we execute it?
  - the value `4` gets stored in `a`
  - the expression `2+a` is evaluated, resulting in the value 6
  - `6` is passed into the `float` function

Putting it all together

- Consider this program:

```
a = 4
b = 6.0
```

- What happens when we execute it?
    - the value `4` gets stored in `a`
    - the expression `2+a` is evaluated, resulting in the value 6
    - `6` is passed into the `float` function
    - the `float` function converts 6 to a `float` and returns `6.0`

Putting it all together

- Consider this program:

```
a = 4
b = 6.0
```

- What happens when we execute it?
  - the value `4` gets stored in `a`
  - the expression `2+a` is evaluated, resulting in the value 6
  - `6` is passed into the `float` function
  - the `float` function converts 6 to a `float` and returns `6.0`
  - the value `6.0` gets stored in variable `b`

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
```

Putting it all together

- In what order do things get evaluated?
- A function’s arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))

print(4, 4+6, int(10.4))
```

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
print(4, 4+6, int(10.4))
print(4, 10, int(10.4))
```

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
print(4, 4+6, int(10.4))
print(4, 10, int(10.4))
print(4, 10, 10)
```

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
print(4, 4+6, int(10.4))
print(4, 10, int(10.4))
print(4, 10, 10)
```

`4 10 10` is printed to the console

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
```

- Parenthesized expressions are evaluated inside-out:

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
```

- Parenthesized expressions are evaluated inside-out: `20 // (6 + 3)`

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
```

- Parenthesized expressions are evaluated inside-out:

```
20 // (6 + 3)

20 // 9
```

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated left-to-right before it is called:

```
print(2+2, 4+6, int(10.4))
```

- Parenthesized expressions are evaluated inside-out:

```
     20 // (6 + 3)

     20 // 9

  => 2
```

Putting it all together

- In what order do things get evaluated?
- A function's arguments are always evaluated before it is called

```
print(2+2, 4+6, int(10.4))
```

- Parenthesized expressions are evaluated inside-out: `20 // (6 + 3)`
- More next time on *operator precedence*

Try it out...

What does the following program print?

```
a = 31
b = a // 4
c = (5 % b) - 1.0
print("a", a ** 0, sep=": ", end=";   ")
print("b", b - 4, sep=": ", end=";   ")
print("c", c * 2, sep=": ")
```