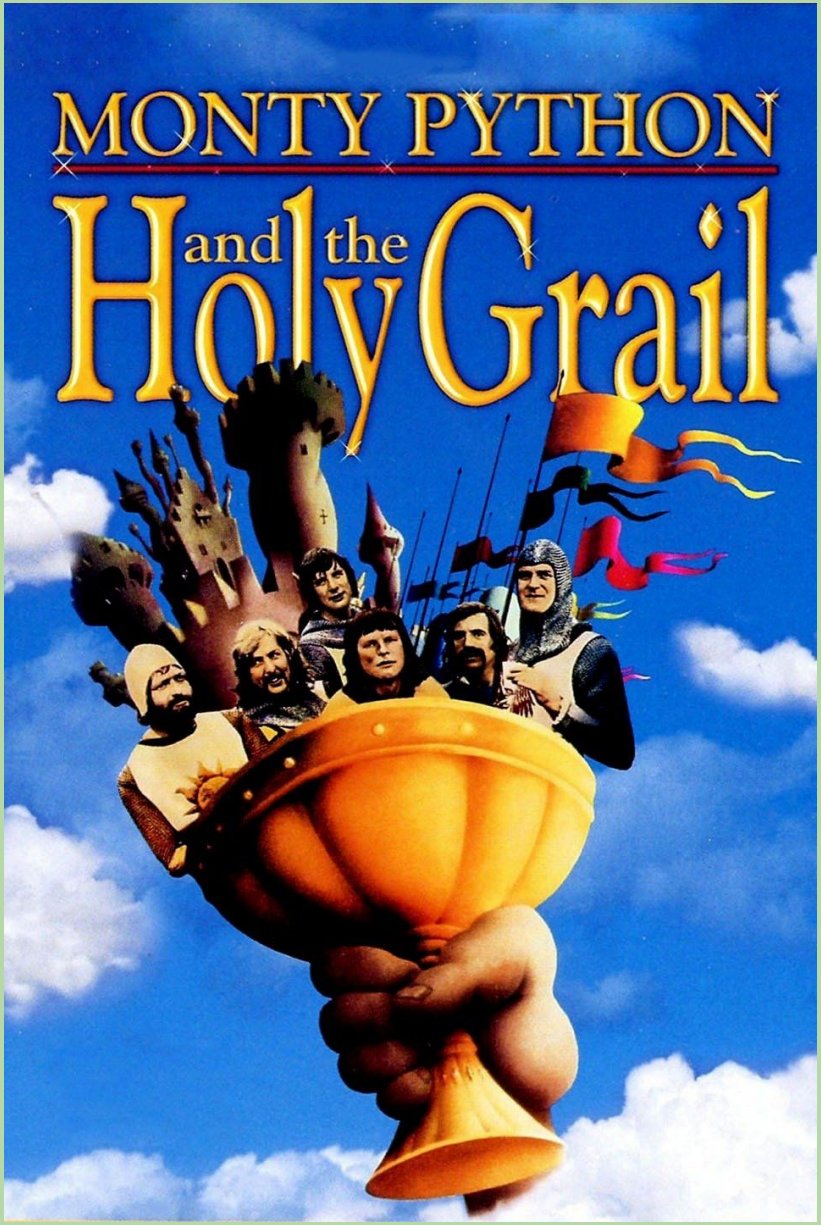
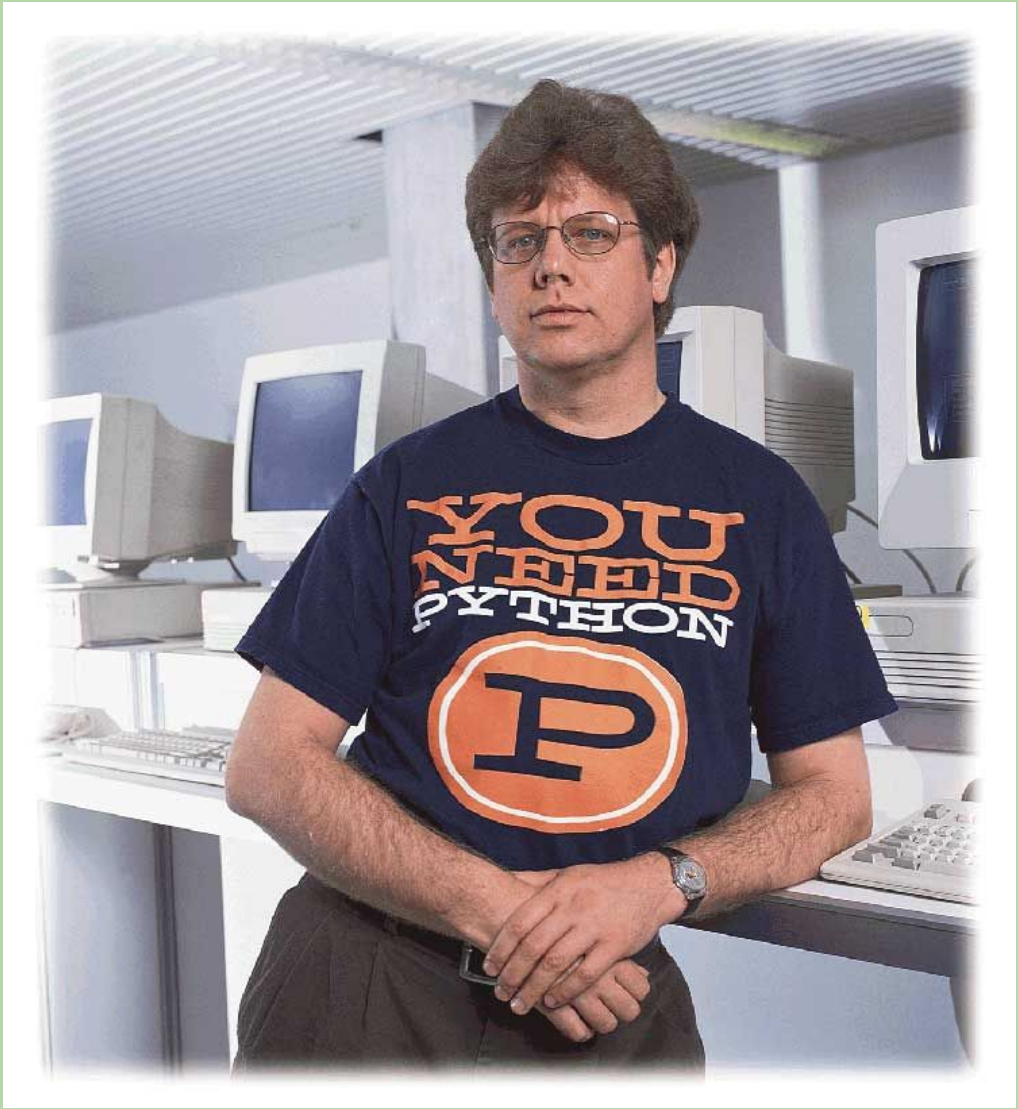
Lecture 01
Introduction to Python

Announcements

- First homework due Wednesday at 4 PM
- You should have signed up for:
  - weekly TA section
  - Piazza
  - REEF for iClickers
  - signed the collaboration policy agreement
- If you haven't, please do so ASAP! If you're having trouble, please email the HTAs
- Check out our instagram account @cs4thefans
- Don't forget to use Piazza and TA Hours as resources!!

Python!





Guido van Rossum: creator of Python

iClicker check

What country is Guido von Rossum originally from?

A. **Canada**
B. **Germany**
C. **Holland**
D. **Netherlands**
E. **United States**

iClicker check

What country is Guido von Rossum originally from?

A. **Canada**
B. **Germany**
C. **Holland ⇐ not a country!**
D. **Netherlands**
E. **United States**

Interacting with Python

- We're using Python **3** (*not* 2).
  - see course website for how to install

- When you start Python, you get the Python *Shell.*

- The following prompt indicates that the Shell is waiting for you to type something:

```
>>>
```

Arithmetic in Python

- Numeric operators include:
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `**` exponentiation
  - `%` modulus: gives the remainder of a division

Data Types and Operators

- There are really two sets of numeric operators:
  - one for integers (ints)
  - one for floating-point numbers (floats)

- In most cases, the following rules apply:
  - if *at least one* of the operands is a float, the result is a float
  - if *both* of the operands are ints, the result is an int

- One exception: division!

Arithmetic in Python (cont.)

The operators follow the PEMDAS order of operations (almost)

Exceptions:

- Multiplication and Division are evaluated left to right
- Addition and Subtraction are evaluated left to right

Recall PEMDAS!

```
>>> 2 / 2 + 1 * 3
4.0

>>> 2 / (2 + 1) * 3
2.0
```

Use parentheses to avoid confusion!

Numeric Data Types

- Different kinds of values are stored and manipulated differently.
- Python *data types* include:
  - integers
    - example: `451`
  - floating-point numbers
    - numbers that can include a decimal (fractional part)
    - example: `3.1416`

Two Types of Division

- The / operator *always* produces a float result.
  - examples:

```
>>> 5 / 3
1.6666666666666667

>>> 6 / 3
2.0
```

Two Types of Division (cont.)

- There is a separate `//` operator for *integer* division.

```
>>> 6 // 3
2
```

- Integer division *discards* any fractional part of the result:

```
>>> 11 // 5
2
>>> 5 // 3
1
```

- Note that it does *not* round!
  - i.e. only the "whole part" of the division and not the fractional part is returned ("floor" function or "truncation")

Another Data Type

- A *string* is a sequence of characters/symbols
  - surrounded by single or double quotes
  - examples:

```
“Hello"
'Picobot'
“Your mother was a hamster, and your father
smelt of elderberries.”
```

Variables

- Variables allow us to store a value for later use:

```
>>> temp = 77
>>> temp - 5
72
>>> (temp - 32) * 5 / 9
25.0
```

- Updating a variable requires assignment to a new value

```
>>> temp = 80
>>> temp
80
```

Expressions

- *Expressions* produce a value.
  - Python *evaluates* them to obtain their value.

- They include:
  - *literals* ("hard-coded" values):

    ```
    3.1416
    'Picobot'
    ```

  - variables

    ```
    temp
    ```

  - combinations of literals, variables, and operators:

    ```
    (temp - 32) * 5 / 9
    ```

Evaluating Expressions with Variables

- When an expression includes variables, they are first replaced with their current value.
- Example (showing how Python would evaluate this):

```
(temp - 32) * 5 / 9
( 77  - 32) * 5 / 9
        45 * 5 / 9
        225 / 9
           25.0
```

Statements

- A *statement* is a command that carries out an action.

- A *program* is a sequence of statements.

```
quarters = 2
dimes = 3
nickels = 1
pennies = 4
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print('you have', cents, 'cents')
```

Assignment Statements

- *Assignment statements* store a value in a variable.

```
temp = 20
```

= is known as the *assignment operator*

- General syntax:

variable = *expression*

- Steps:
  1) evaluate the expression on the right-hand side of the =
  2) assign the resulting value to the variable on the left-hand side of the =

- Example:

```
quarters = 10

quarters_val = 25 * quarters
quarters_val = 25 * 10
quarters_val = 250
```

Assignment Statements (cont.)

- We can change the value of a variable by assigning it a new value.

- Example:

Fill in the blanks!

`num1 = 100` `num2 = 120`	num1 [100] num2 [120]
`num1 = 50`	num1 [ ] num2 [ ]
`num1 = num2 * 2`	num1 [ ] num2 [ ]
`num2 = 60`	num1 [ ] num2 [ ]

Assignment Statements (cont.)

- We can change the value of a variable by assigning it a new value.

- Example:

num1 = 100 num2 = 120	num1 100 num2 120
num1 = 50	num1 50 num2 120
num1 = num2 * 2 120 * 2 240	num1 240 num2 120
num2 = 60	num1 240 num2 60

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!

- Example:

Fill in the blanks!

`sum = 13` `val = 30`	`sum` 13 `val` 30
`sum = sum + val`	`sum` `val`
`val = val * 2`	`sum` `val`

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!

- Example:

`sum = 13` `val = 30`	

`sum = sum + val`
`13 + 30`
`43`
	

`val = val * 2`
`30 * 2`
`60`
	

Creating a Reusable Program

- Put the statements in a text file.

```
# a program to compute the value of some coins

quarters = 2        # number of quarters
dimes = 3
nickels = 1
pennies = 4

cents = quarters*25 + dimes*10 + nickels*5 + pennies
print('you have', cents, 'cents')
```

- Program file names should have the extension `.py`
  - example: `coins.py`

Print Statements

- print statements display one or more values on the screen

- Basic syntax:

  `print(`*expr*`)`

  or

  `print(`$expr_1$`, `$expr_2$`, ... `$expr_n$`)`

  where each *expr* is an expression

- Steps taken when executed:
  1. the individual expression(s) are evaluated
  2. the resulting values are displayed on the same line, *separated by spaces*

- To print a blank line, omit the expressions:

  ```
  print()
  ```

Print Statements (cont.)

- Examples:
  - first example:

```
print('the results are:', 15 + 5, 15 - 5)
```

```
'the results are:'    20      10
```

output: `the results are: 20 10`

(note that the quotes around the string literal are *not* printed)

  - second example:

```
cents = 89
print('you have', cents, 'cents')
```

```
'you have'    89    'cents'
```

output: `you have 89 cents`

Variables and Data Types

- The `type` function gives us the type of an expression:

```
>>> type('hello')
<class 'str'>
>>> type(5 / 2)
<class 'float'>
```

- Variables in Python do *not* have a fixed type.
  - examples:

```
>>> temp = 25.0
>>> type(temp)
<class 'float'>
>>> temp = 77
>>> type(temp)
<class 'int'>
```

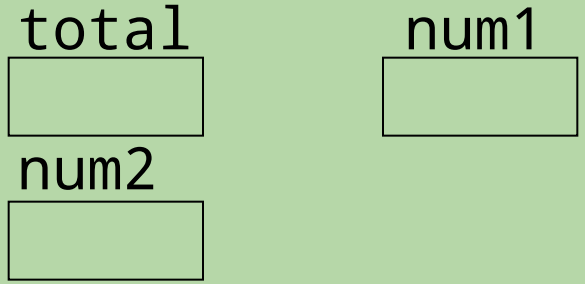
How a Program Flows...

- Flow of control = order in which statements are executed
- By default, a program's statements are executed sequentially, from top to bottom.

example program

```
total = 0
num1 = 5
num2 = 10
total = num1 + num2
```

variables in memory



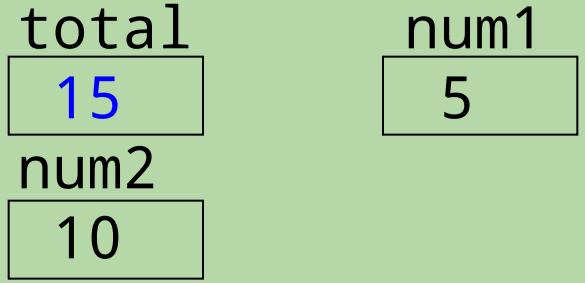
How a Program Flows...

- Flow of control = order in which statements are executed
- By default, a program's statements are executed sequentially, from top to bottom.

example program

```
total = 0
num1 = 5
num2 = 10
total = num1 + num2
total = num1 + num2
          5  +  10
            15
```

variables in memory



What is the output of the following program?

```
x = 15
name = 'Picobot'
x = x // 2
print('name ', x, type(x))
```

A. Picobot 7 <class 'int'>

B. Picobot 7.5 <class 'float'>

C. name 8 <class 'int'>

D. name 7 <class 'int'>

E. name 7.5 <class 'float'>

What is the output of the following program?

```
x = 15
name = 'Picobot'
x = x // 2                                  # x = x // 2
print('name ', x, type(x))                        15 // 2
                                                   7
        'name '  7  type(7)
                    <class 'int'>
```

A. Picobot 7 <class 'int'>

B. Picobot 7.5 <class 'float'>

C. name 8 <class 'int'>

D. **name 7 <class 'int'>**

E. name 7.5 <class 'float'>

Extra Practice: What about this program?

```
x = 15
name = 'Picobot'
x = 7.5
print(name, ' x ', type(x))
```

A. `name x <class 'float'>`

B. `Picobot 7.5 <class 'float'>`

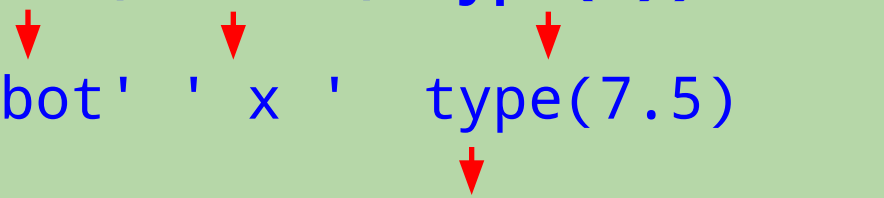
C. `Picobot x <class 'float'>`

D. `Picobot 15 <class 'int'>`

E. `name 7.5 <class 'str'>`

Extra Practice: What about this program?

```
x = 15
name = 'Picobot'
x = 7.5
print(name, ' x ', type(x))
```



A. name x <class 'float'>

B. Picobot 7.5 <class 'float'>

C. **Picobot x <class 'float'>**

D. Picobot 15 <class 'int'>

E. name 7.5 <class 'str'>

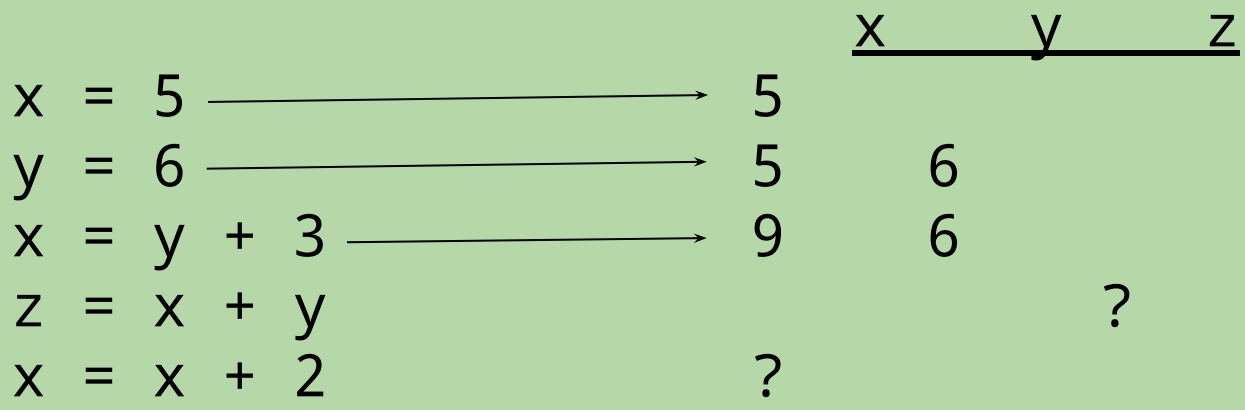
What are the values of the variables after the following code runs?

```
x = 5
y = 6
x = y + 3
z = x + y
x = x + 2
```

	x	y	z
A.	11	6	15
B.	11	6	11
C.	11	6	17
D.	7	6	11

E. none of these, because the code has an error

Hint: create a table of program state changes



	x	y	z
A.	11	6	15
B.	11	6	11
C.	11	6	17
D.	7	6	11

E. none of these, because the code has an error

What are the values of the variables after the following code runs?

```
x = 5
y = 6
x = y + 3
z = x + y
x = x + 2
```

```
    9 + 2
    11
```

x	y	z
5		
5	6	
9	6	
9	6	15
11	6	15

	x	y	z
A.	**11**	**6**	**15**
B.	11	6	11
C.	11	6	17
D.	7	6	11

E. none of these, because the code has an error

changing the value of x
does *not* change the value of z!

Strings: Numbering the Characters

- The position of a character within a string is known as its *index*.
- There are two ways of numbering characters in Python:
  - from left to right, starting from 0

    0 1 2 3 **4**

    'Perry'
  - from right to left, starting from -1

    -5 -4 -3 -2 **-1**

    'Perry'
  - `'P'` has an index of 0 or -5
  - `'y'` has an index of 4 or -1

String Operations

- Indexing: *string*[*index*]

```
>>> name = 'Picobot'
>>> name[1]
'i'
>>> name[-3]
'b'
```

- Slicing (extracting a substring): *string*[*start*:*end*]

```
>>> name[0:2]
'Pi'
>>> name[1:-1]
'icobo'
>>> name[1:]
'icobot'
>>> name[:4]
'Pico'
```

from this index

up to but ***not including*** this index

String Operations (cont.)

- Concatenation: *string1 + string2*

```
>>> word = 'program'
>>> plural = word + 's'
>>> plural
'programs'
```

- Duplication: *string * num_copies*

```
>>> 'ho!' * 3
'ho!ho!ho!'
```

- Determining the length: `len(`*string*`)`

```
>>> name = 'Perry'
>>> len(name)
5
>>> len('')   # an empty string – no characters!
0
```

String Operations (cont.)

- Concatenation: *string1 + string2*

```
>>> word = 'program'
>>> plural = word + 's'
>>> plural
'programs'
```

- Duplication: *string * num_copies*

```
>>> 'ho!' * 3
'ho!ho!ho!'
```

- Determining the length: `len(`*string*`)`

```
>>> name = 'Perry'
>>> len(name)
5
>>> len('')   # an empty string – no characters!
0
```

Remark:
Operators depends on the types of their operands

<type 'str'> + <type 'str'> => concatenation

<type 'str'> * <type 'int'> => duplication

What is the value of `s` after the following code runs?

```
s = 'abc'

s = ('d' * 3) + s

s = s[2:-2]
```

A. `'ddab'`
B. `'dab'`
C. `'dda'`
D. `'da'`
E. none of these

What is the value of s after the following code runs?

```
s = 'abc'

s = ('d' * 3) + s
      'ddd'   + 'abc'   ➔   'dddabc'

s = s[2:-2]
    'dddabc'[2:-2]
```

A. 'ddab'

B. 'dab'

C. 'dda'

D. **'da'**

E. none of these

Skip-Slicing

- Slices can have a third number: *string* [*start* : *end* : *stride_length*]


```
>>> s[0:8:2]
'BonU'          # Note ends at U, not i
```

Skip-Slicing

- Slices can have a third number: *string* [*start* : *end* : *stride_length*]


```
>>> s[0:8:2]
'BonU'          # Note ends at U, not i

>>> s[5:0:-1]
' nwor'         # Note space at beginning
```

Skip-Slicing

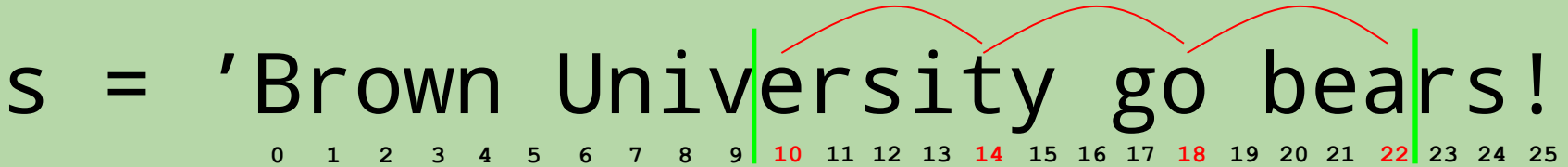
- Slices can have a third number: *string* [*start* : *end* : *stride_length*]

```
s = 'Brown University go bears!'
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
>>> s[0:8:2]
'BonU'          # Note ends at U, not i

>>> s[5:0:-1]
' nwor'         # Note space at beginning

>>> s[ : : ]  # what numbers do we need?
'etoa'

>>> s[0::23]+s[6:0:-2]+s[-1]*2  # what do we get?
```

Skip-Slicing

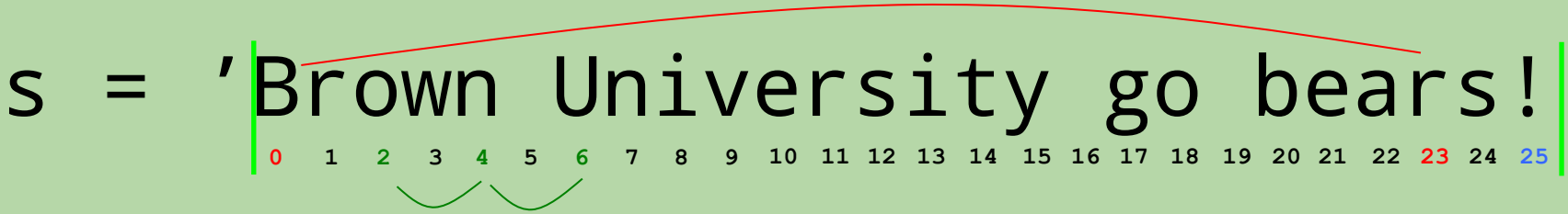
- Slices can have a third number: *string* [*start* : *end* : *stride_length*]




```
>>> s[0:8:2]
'BonU'          # Note ends at U, not i

>>> s[5:0:-1]
' nwor'         # Note space at beginning

>>> s[10:23:4]  # or s[10::4] or ...
'etoa'

>>> s[0::23]+s[6:0:-2]+s[-1]*2  # what do we get?
```

Skip-Slicing

- Slices can have a third number: *string* [*start* : *end* : *stride_length*]




```
>>> s[0:8:2]
'BonU'          # Note ends at U, not i

>>> s[5:0:-1]
' nwor'         # Note space at beginning

>>> s[10:23:4]  # or s[10::4] or ...
'etoa'

>>> s[0::23]+s[6:0:-2]+s[-1]*2  # what do we get?
'BrUno!!'
```

Lists

Lists

- A string is a sequence of characters.

  'hello'

- A list is a sequence of *arbitrary* values (the list's *elements*).

  [2, 4, 6, 8]

  ['CS', 'math', 'english', 'psych']

- A list can include values of different types:

  ['Star Wars', 1977, 'PG', [35.9, 460.9]]

List Ops == String Ops (more or less)

```
                    0       1           2           3
>>> majors = ['CS', 'math', 'english', 'psych']

>>> majors[2]
'english'

>>> majors[1:3]
['math', 'english']

>>> len(majors)
4

>>> majors + ['physics']
['CS', 'math', 'english', 'psych', 'physics']

>>> majors[::-2]
???
```

List Ops == String Ops (more or less)

```
>>> majors = ['CS', 'math', 'english', 'psych']

>>> majors[2]
'english'

>>> majors[1:3]
['math', 'english']

>>> len(majors)
4

>>> majors + ['physics']
['CS', 'math', 'english', 'psych', 'physics']

>>> majors[::-2]
['psych', 'math']
```

What is the output of the following program?

```
mylist = [1, 2, [3, 4, 5]]
print(mylist[1], mylist[1:2])
```

A. `2 2 3`

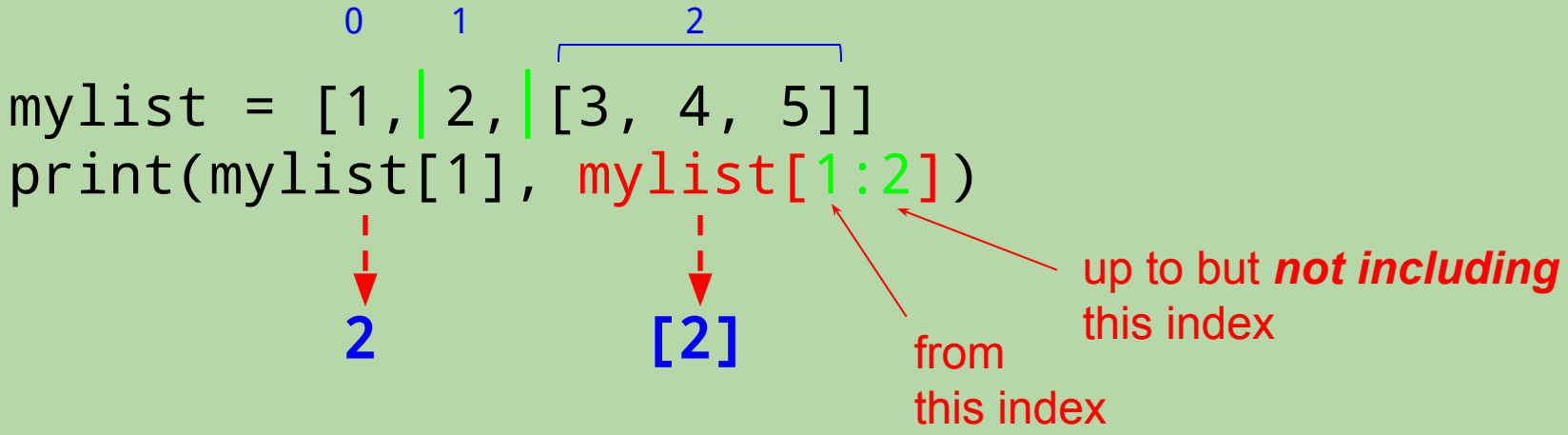
B. `2 [2, 3]`

C. `2 2`

D. `2 2 [3, 4, 5]`

E. none of these

What is the output of the following program?



A. `2 2 3`

B. `2 [2, 3]`

C. `2 2`

D. `2 2 [3, 4, 5]`

E. **none of these!!** **`2 [2]`**

Slicing a list always produces a list!

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> ???  # what is needed?
35.9
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> info[-1][0]
35.9
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> info[-1][0]
35.9
>>> info[-1][-1]
???
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> info[-1][0]
35.9
>>> info[-1][-1]
460.9
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> info[-1][0]
35.9
>>> info[-1][-1]
460.9
>>> info[0][-4]
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Bears'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> info[-1][0]
35.9
>>> info[-1][-1]
460.9
>>> info[0][-4]
'W'
```

How could you fill in the blank to produce [105, 111]?

```
intro_cs = [101, 103, 105, 108, 109, 111]

new_courses = _____________________________
```

A. `intro_cs[2:3] + intro_cs[-1:]`

B. `intro_cs[-4] + intro_cs[5]`

C. `intro_cs[-4] + intro_cs[-1:]`

D. more than one of the above

E. none of the above

How could you fill in the blank to produce [105, 111]?

```
                0     1     2     3     4     5
intro_cs = [101, 103, 105, 108, 109, 111]
              -6    -5    -4    -3    -2    -1
new_courses = _____________________________
```

A. **`intro_cs[2:3] + intro_cs[-1:]`**
   `[105] + [111] → [105, 111]`

B. `intro_cs[-4] + intro_cs[5]`
   `105 + 111 → 216`

C. `intro_cs[-4] + intro_cs[-1:]`
   `105 + [111] → error!`

D. more than one of the above

E. none of the above

Extra Practice: Fill in the blank to make the code print `'compute!'`

```
subject = 'computer science!'
verb = _________________
print(verb)
```

A. `subject[:7] + subject[-1]`

B. `subject[:7] + subject[:-1]`

C. `subject[:8] + subject[-1]`

D. `subject[:8] + subject[:-1]`

E. none of these

Extra Practice: Fill in the blank to make the code print `'compute!'`

```
subject = 'computer science!'
verb = _________________
print(verb)
```

A. `subject[:7] + subject[-1]`

B. `subject[:7] + subject[:-1]`

C. `subject[:8] + subject[-1]`

D. `subject[:8] + subject[:-1]`

E. none of these

Extra practice from the textbook authors!

```
pi = [3,1,4,1,5,9]
L = [ 'pi', "isn't", [4,2] ]
M = 'You need parentheses for chemistry !'
    0       4       8       12      16      20      24      28      32
```

Part 1

What is `len(pi)`

What is `len(L)`

What is `len(L[1])`

What is `pi[2:4]`

What slice of `pi` is `[3,1,4]`

What slice of `pi` is `[3,4,5]`

Part 2

What is `L[0]`

These two are different!

What is `L[0:1]`

What is `L[0][1]`

What slice of `M` is `'try'`?

is `'shoe'`?

What is `M[9:15]`

What is `M[::5]`

Extra!

What are `pi[0]*(pi[1] + pi[2])` and `pi[0]*(pi[1:2] + pi[2:3])` ?

These two are different, too…

Extra practice from the textbook authors!

```
pi = [3,1,4,1,5,9]
L = [ 'pi', "isn't", [4,2] ]
M = 'You need parentheses for chemistry !'
    0   4   8   12  16  20  24  28  32
```

Part 1

What is `len(pi)` 6

What is `len(L)` 3

What is `len(L[1])` 5

What is `pi[2:4]` [4, 1]

What slice of `pi` is `[3,1,4]` pi[:3]

What slice of `pi` is `[3,4,5]` pi[::2]

Part 2

What is `L[0]` 'pi'

These two are different!

What is `L[0:1]` ['pi']

What is `L[0][1]` 'i'

What slice of `M` is `'try'`? M[31:34]

is `'shoe'`? M[30:17:-4]

What is `M[9:15]` 'parent'

What is `M[::5]` 'Yeah cs!'

Extra!

What are `pi[0]*(pi[1] + pi[2])` and `pi[0]*(pi[1:2] + pi[2:3])` ?

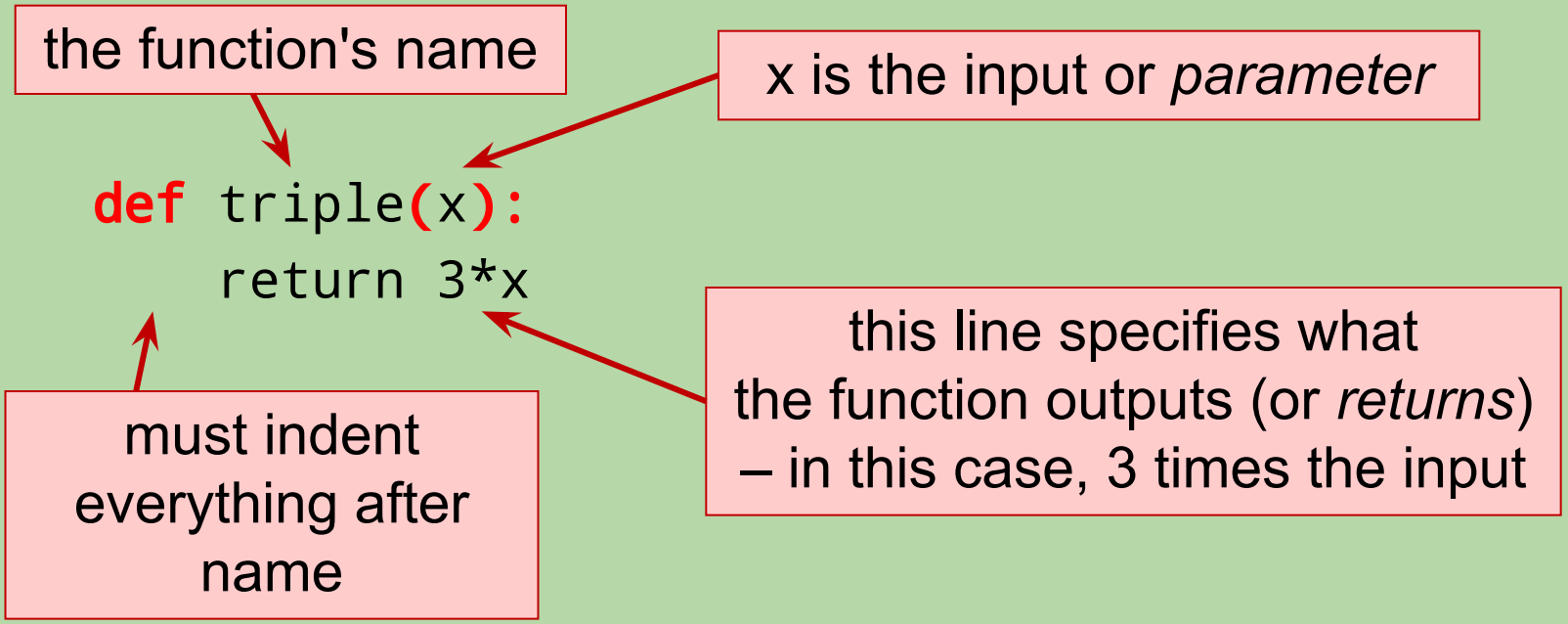
These two are different, too…

15

[1, 4, 1, 4, 1, 4]

Functions

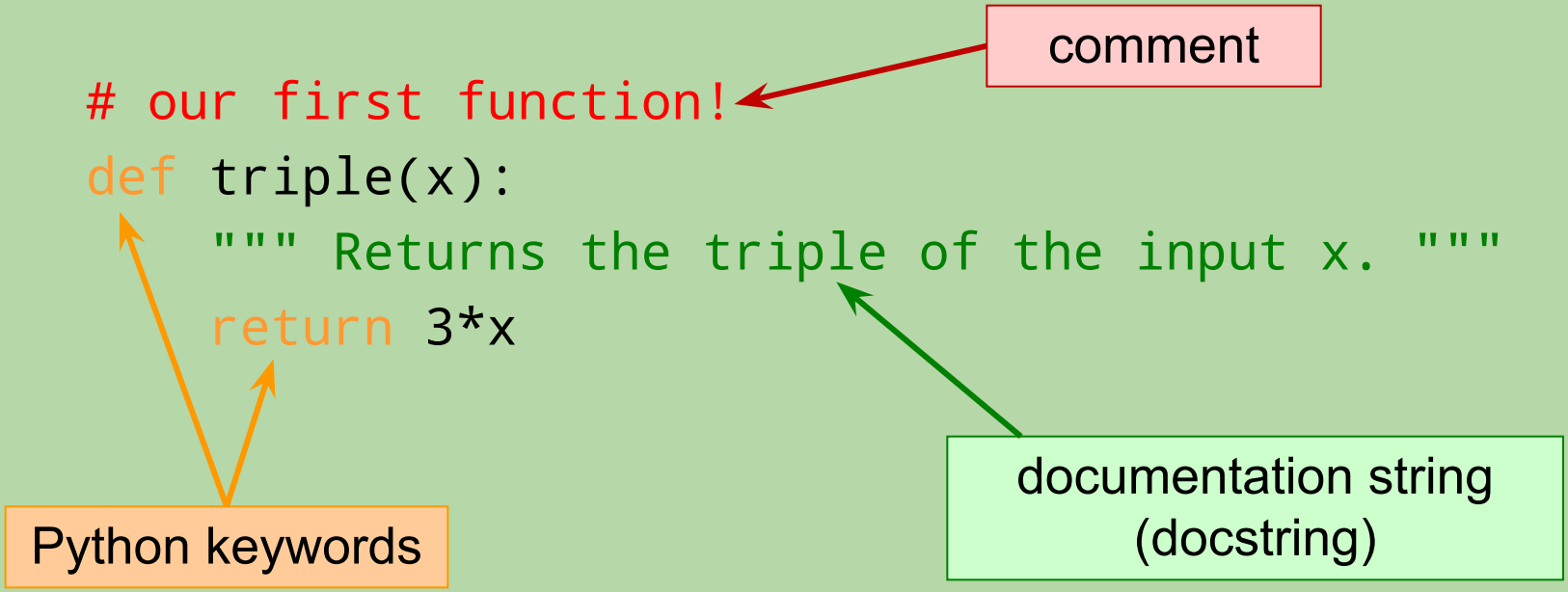
Defining a Function



- Once we define a function, we can call it:

```
>>> triple(3)
9
>>> triple(10)
30
>>> triple(0.5)
1.5
```

Other Details



- Python uses color-coding to distinguish program components.
- Always use a *docstring* to explain what the function does.
  - surrounded by triple quotes, beginning on the second line
  - `help(`*function name*`)` retrieves it
- Other (non-docstring) comments can be included as needed.

Functions With String Inputs

```
def undo(s):
    """ Adds the prefix "un" to the input s. """
    return 'un' + s

def redo(s):
    """ Adds the prefix "re" to the input s. """
    return 're' + s
```

- Examples:

```
>>> undo('plugged')
'unplugged'
>>> undo('zipped')
'unzipped'
>>> redo('submit')
???
>>> redo(undo('zipped'))
???
```

The evil "un" people!
(from the PBS kids show *Between the Lions*)

Functions With String Inputs

```
def undo(s):
    """ Adds the prefix "un" to the input s. """
    return 'un' + s

def redo(s):
    """ Adds the prefix "re" to the input s. """
    return 're' + s
```

- Examples:

```
>>> undo('plugged')
'unplugged'
>>> undo('zipped')
'unzipped'
>>> redo('submit')
'resubmit'
>>> redo(undo('zipped'))      # redo('unzipped')
'reunzipped'
```

The evil "un" people!
(from the PBS kids show *Between the Lions*)

Multiple Lines, Multiple Parameters

```
def circle_area(diam):
    """ Computes the area of a circle
        with a diameter diam.
    """
    radius = diam / 2
    area = 3.14159 * (radius**2)
    return area

def rect_perim(l, w):
    """ Computes the perimeter of a rectangle
        with length l and width w.
    """
    return 2*l + 2*w
```

- Examples:

```
>>> rect_perim(5, 7)
24
>>> circle_area(20)
314.159
```

Function and Function Call in the Same File

```
def circle_area(diam):
    """ Computes the area of a circle
        with a diameter diam.
    """
    radius = diam / 2
    area = 3.14159 * (radius**2)
    return area

def rect_perim(l, w):
    """ Computes the perimeter of a rectangle
        with length l and width w.
    """
    return 2*l + 2*w

print(rect_perim(20, 8))    # Why is print needed?
```

- Defines two functions, but only one gets called when we run the program.
- We can still call either of them from the Shell after running the program.

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a * b - 3


print(calculate(3, 2))
```

A. 5

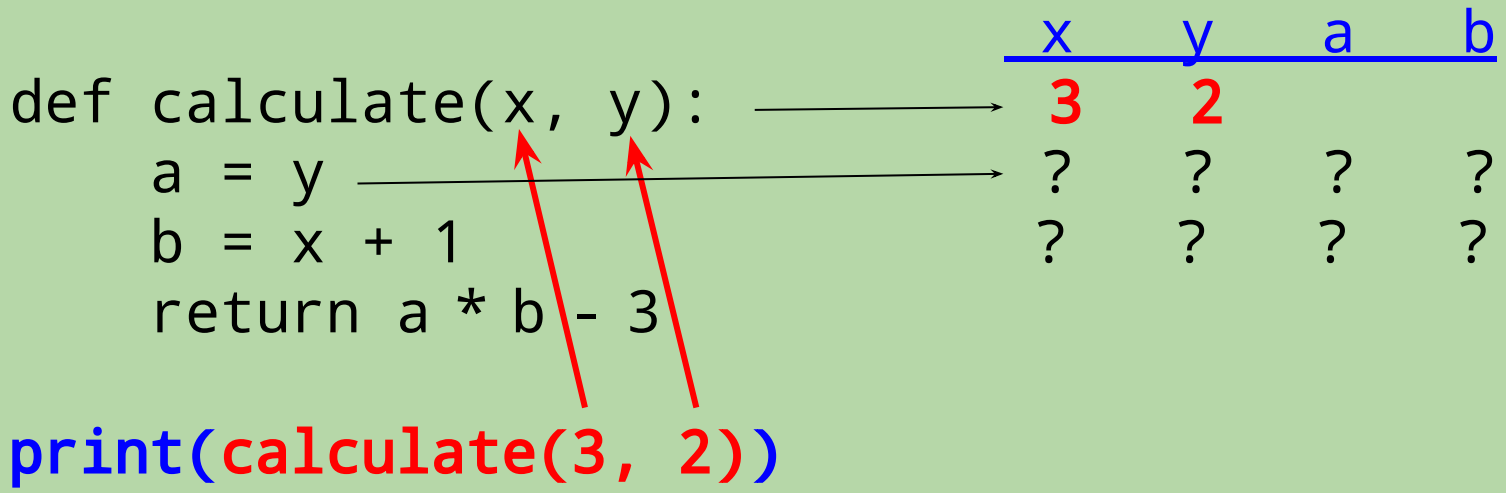
B. 9

C. 4

D. 3

E. 8

What is the output of this code?



A. 5

B. 9

C. 4

D. 3

E. 8

The values in the function call are assigned to the parameters.

In this case, it's as if we had written:

```
x = 3
y = 2
```

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a * b - 3
```

x	y	a	b
3	2		
		2	
			4

2 * 4 - 3 = 5

```
print(calculate(3, 2))    # print(5)
```

A. **5**

B. 9

C. 4

D. 3

E. 8

The output/return value:

- is sent back to where the function call was made
- replaces the function call

The program picks up where it left off when the function call was made.

Practice Writing a Function

- Write a function `middle_char(s)` that takes a string `s` with at least one character, and returns the middle character in `s`

```
>>> middle_char('alien')
'i'
>>> middle_char('function')
't'
```

```
def middle_char(s):
        middle_index = _______________
        return _________________
```

Practice Writing a Function

- Write a function `middle_char(s)` that takes a string `s` with at least one character, and returns the middle character in `s`

```
>>> middle_char('alien')
'i'
>>> middle_char('function')
't'
```

```
def middle_char(s):
        middle_index = len(s) // 2
        return s[middle_index]
```