Functions

CSCI 1030U - Intro to Computer Science
@IntroCS

Randy J. Fortier
@randy_fortier

Outline

- Functions
- Calling functions
  - Argument passing
    - Pass by value
    - Pass by reference

Functions

Modularity

- So far, we've created only small programs
  - When programs get large, they become more complex to write, debug, modify, and understand
  - Modularity can make it easier to comprehend large programs
  - Modularity can also make it possible to reuse part of our program
  - Modules can be tested separately (unit testing)
- Types of modularity:
  - Functions
  - Objects (discussed later)

Functions

- A function is a module of program code
  - A function takes input (arguments)
    - The arguments allow us to customize the operation performed by the function
  - A function produces output (return value)
    - The return value is often the *result* of executing the code

Functions - Syntax

- Functions without arguments or return value:

Python:

```python
def say_hello():
    print("Hello!")
```

C++:

```cpp
void sayHello() {
    cout << "Hello!";
}
```

- To call this function:

Python:

```python
say_hello()
```

C++:

```cpp
sayHello();
```

Functions - Syntax

- Functions with a return value:

Python:

```python
def get_answer():
    return 42
```

C++:

```cpp
int getAnswer() {
    return 42;
}
```

- To call this function:

Python:

```python
answer = get_answer()
```

C++:

```cpp
int answer = getAnswer();
```

Functions - Syntax

- Functions with arguments:

Python:

```python
def get_dog_age(h_age):
    return h_age * 7
```

C++:

```cpp
int getDogAge(int hAge) {
    return hAge * 7;
}
```

- To call this function:

Python:

```python
dog_age = get_dog_age(24)
```

C++:

```cpp
int dAge = getDogAge(24);
```

Functions - Documentation

- To document a function, use a multi-line comment immediately after the `def` line:

```
def get_age_in_dog_years(human_age):
    """
    This function, given an age in human
    years, returns the age in dog years.
    """
    return human_age * 7
```

Local Variables

- If you use any variables inside functions, they are *local variables*
  - A local variable is accessible/usable within that function only
  - The word local refers to the variable's *scope*
    - The scope is local to the function

```
def get_age_in_dog_years(human_age):
    dog_years_factor = 7  # local variable
    return human_age * dog_years_factor
```

Global Variables

- Variables used outside of a function are called *global variables*
  - Global variables' scope includes both inside and outside of functions
  - However, since there could be naming conflicts, we have to explicitly declare when we use global variables
  - Generally, using global variables like this is a bad idea

```
dog_years_factor = 7  # global variable
def get_age_in_dog_years(human_age):
   global dog_years_factor
   return human_age * dog_years_factor
```

Note: It is recommended to avoid using global variables within functions.

Function Calling

OntarioTech UNIVERSITY

Function Calling - Argument Passing

- Consider the following situation:

```
def random(low, high):
    return random.randint(low, high)

max = 10
print(random(0, max))
```

- The number 10 seems to have two different names: `high` and `max`

Argument Type Hints

- Even though Python is dynamically typed, you can specify type hints to help users know how to use your functions:

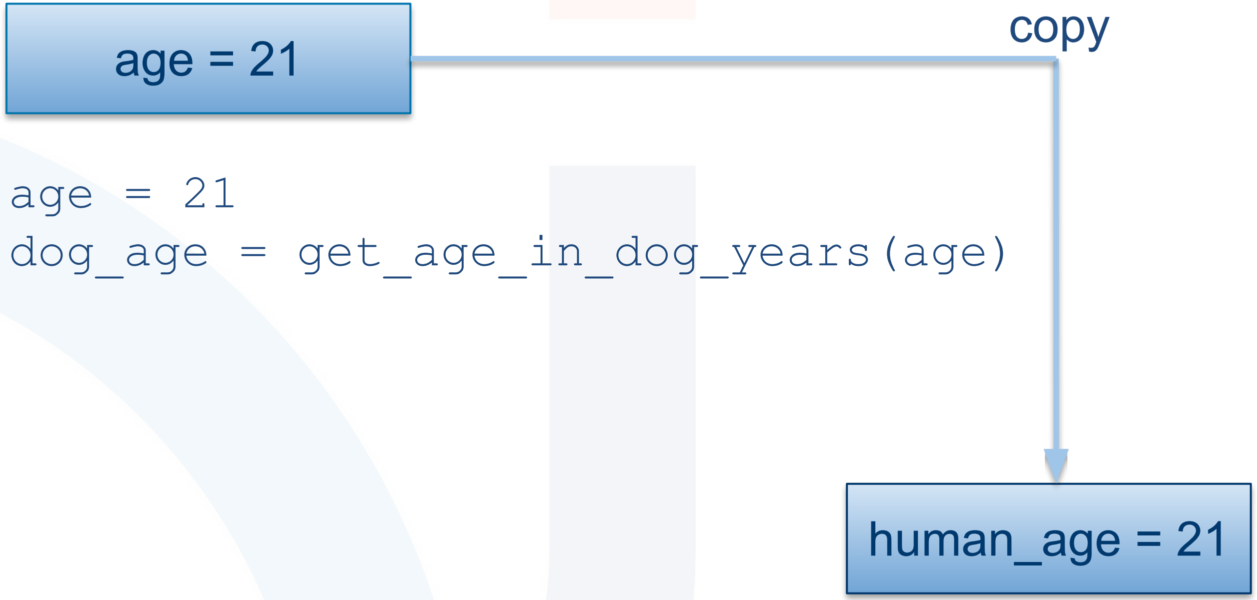
```
def random(low: int, high: int) -> int:
    return random.randint(low, high)
```

Function Calling - Pass by Value

- One way that arguments are passed into a function is *by value*
  - By value means that the value passed to the function when it is called is copied, and the copy is put into the argument variable
    - An argument variable has the same scope as a local variable
- Advantage:
  - When calling a function by passing its arguments from variables, you don't have to worry about those variables' values being modified

OntarioTech UNIVERSITY

Function Calling - Pass by Value



```
age = 21
dog_age = get_age_in_dog_years(age)
```

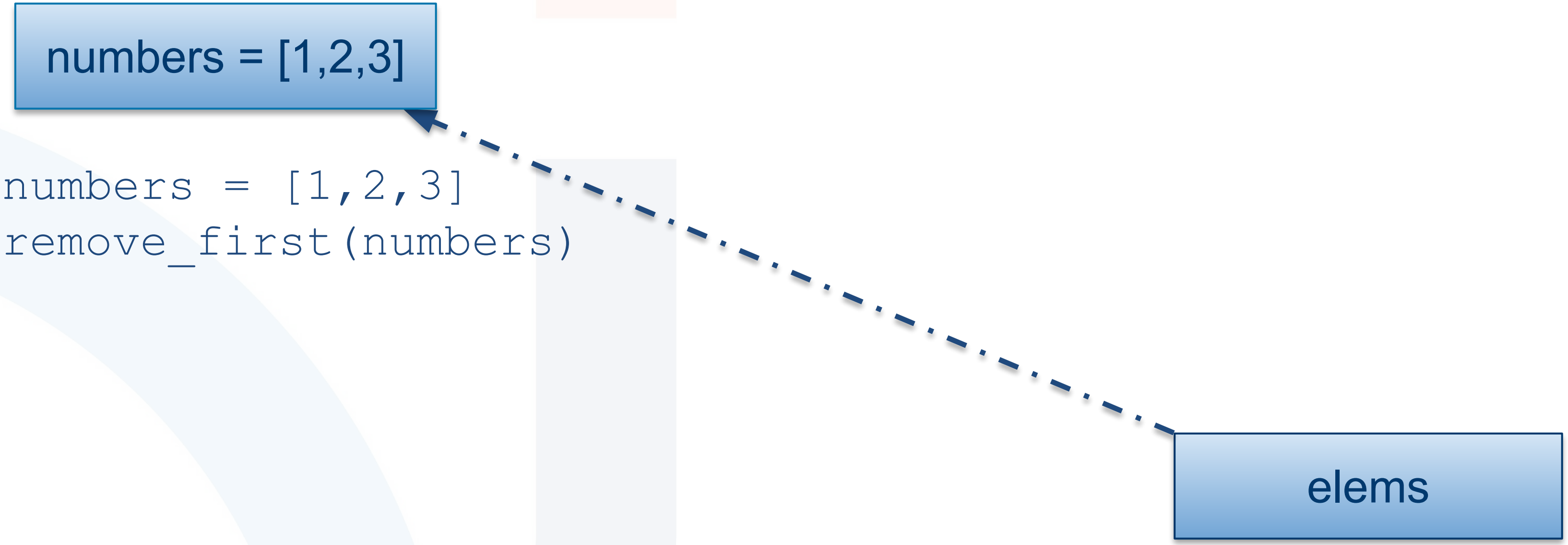
```
def get_age_in_dog_years(human_age):
    return human_age * 7
```

Function Calling - Pass by Reference

- In many programming languages, you can also pass argument values by reference
  - This is possible in C++ with the & operator
  - By reference means that the values become linked via the argument variable
    - In other words, the argument becomes an alias for the value
- Advantages:
  - Copying large data is not necessary
  - You can pass values to functions that you intend to be modified

- *Python passes all object types by reference*
  - *e.g. strings, lists, dictionaries*

Function Calling - Pass by Reference



```
numbers = [1,2,3]
remove_first(numbers)
```

```
def remove_first(elems):
   elems.pop(0)
```

Function Calling - Named Arguments

- Consider the following function:

```
def distance(x1, y1, x2, y2):
    return math.sqrt((x2-x1)**2 + (y2-y1)**2)
```

- This function can be called using the ordering of the arguments:

```
d = distance(0, 0, 3, 4)
```

- The function can also be called using the names of the arguments:

```
d = distance(x1=0, y1=0, x2=3, y2=4)
```

Function Calling - Argument Defaults

- Consider the following function:

```
def calculate_interest(principal=1000, interest_rate=0.035):
    return principal * interest_rate
```

- This function can be called using no arguments:

```
interest = calculate_interest()
```

- The function can also be called one or both of the arguments:

```
interest = calculate_interest(principle=5000)
```

Coding Exercise 04b.1

- Create a function, named `get_class_average`, which takes a list of numbers (`marks`) as its argument, and returns the average/mean of those numbers
- For example:

```
midterm_marks = [57.0, 62.5, 68.0, 74.0, 55.0, 71.0, 94.5, 47.5]
midterm_average = get_class_average(midterm_marks)
print(f'{midterm_average = }') # 66.1875
```

Hacker's Corner: Lambda Expressions

- Alonzo Church developed a notation for describing unnamed functions, called Lambda Calculus in 1936
  - The name comes from the symbol (the Greek letter lambda, λ), used to denote the arguments of those functions
  - Lambda expressions' body, naturally, must be an *expression*
- The following is an anonymous function that takes two arguments, and returns their sum:

```
λx λy · x + y
```

Hacker's Corner: Lambda Expressions

- Most programming languages allow Lambda expressions to be used for quick, anonymous, function definitions
- The following defines a new function, add, using a Lambda expression:

```
add = lambda x,y: x + y
z = add(1,2)
```

Wrap-up

- Functions
- Calling functions
  - Argument passing
    - Pass by value
    - Pass by reference

Coming Up

- Stacks and the Calling Stack
- Higher-order functions
    - Passing functions as arguments to other functions