Introduction to Problem Solving in Python

COSI 10A

Review: Variables

- A **variable** is a piece of the computer's memory that is given a name and type, and can store a value
- Steps for using a variable:
  - **Declare/initialize it** - state its name and type and store a value into it
  - **Use it** - print it or use it as part of an expression

Review: Declaration and Assignment

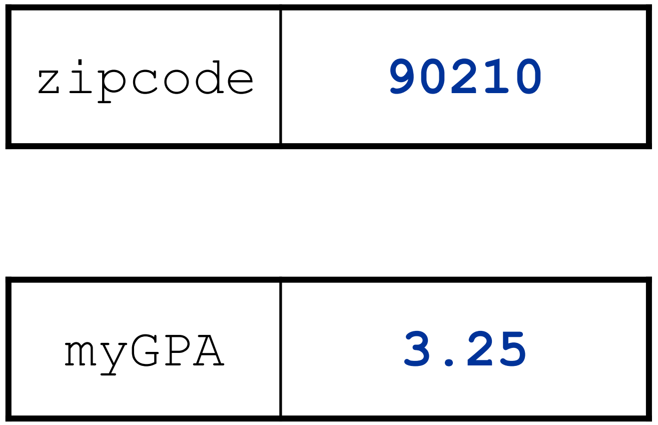
- **Variable declaration and assignment:** Sets aside memory for storing a value and stores a value into a variable
  - Variables **must be declared** before they can be used
  - The value can be an expression. The variable will store its result

- Syntax:

```
name = expression
```

- Example:

```
zipcode = 90210
myGPA = 1.0 + 2.25
```



Class objectives

- Type Conversions
- Python interactive programs
- For Loops

Type conversion functions

Conversion Functions

- Python provides built-in functions that convert from one type to another
- The `int` function takes a compatible value and coverts it to an integer

```
>>> int('32')
32
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> int('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
>>>
```

- It can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part

Conversion Functions

- Python provides built-in functions that convert from one type to another
- The `float` function converts integers and strings to floating-point numbers

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
>>>
```

Conversion Functions

- Python provides built-in functions that convert from one type to another
- The `str` function converts its argument to a string

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
>>>
```

Just in case ...

- If you are not sure what type a value has, the interpreter can tell you

```
>>> type("Hello")
<class 'str'>
>>> type('Hello')
<class 'str'>
>>> type(17)
<class 'int'>
>>> age = 29
>>> type(age)
<class 'int'>
>>>
```

Interactive Programs

Interactive programs

- An **interactive program** reads input from the console
- While the program runs, it asks the user to type input
- The input typed by the user is stored in variables in the code

- It can be tricky; users are unpredictable and misbehave

`input`

- `input` is a function that can read input from the user
- Syntax:

```
name = input(prompt)
```

- Example:

```
myname = input("type your name: ")
```

- The variable `myname` will store the value the user typed in

input example

```
def main():
    age = input("How old are you? ")

    years = 65 - age
    print(years, " years until retirement!")

main()
```

age $\boxed{29}$

- Console window:

```
How old are you?  29
```

Does this look ok?

input example

```
def main():
    age = input("How old are you? ")

    years = 65 - age
    print(years, " years until retirement!")

main()
```

age $\boxed{29}$

- Console window:

```
How old are you?  29
```

```
How old are you? 23
Traceback (most recent call last):
  File "t1.py", line 8, in <module>
    main()
  File "t1.py", line 5, in main
    years = 65 - age
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

`input` example

```
def main():
    age = int(input("How old are you? "))

    years = 65 - age
    print(years, " years until retirement!")

main()
```

age $\boxed{29}$

years $\boxed{36}$

- Console window:

```
How old are you?  29
36 years to retirement!
```

Definite Loops - for loops

Getting rid of repetition

- Functions
- Variables
- What if you want to repeat function calls?

Repetition with `for` loops

- Repeating an action results in redundant code:

```
make_batter()
bake_cookies()
bake_cookies()
bake_cookies()
bake_cookies()
bake_cookies()
frost_cookies()
```

- A `for` loop statement performs a task many times

```
mix_batter()
for i in range(5):     # repeat 5 times
       bake_cookies()
frost_cookies()
```

Control structures

- The `for` loop is an example of looping control structure
- A **control structure** is a program construct that affects the flow of a program's execution
- Controlled code may include one or more statements

Repetition with `for` loops

- Repeating an action results in redundant code:

```
make_batter()
bake_cookies()
bake_cookies()
bake_cookies()
bake_cookies()
bake_cookies()
frost_cookies()
```

- A `for` loop statement performs a task many times

```
mix_batter()
for i in range(1, 6):     # repeat 5 times
      bake_cookies()
frost_cookies
```

Ways to create ranges

Range From	Description	Example	Numbers in Range
`range(max)`	Range from 0 (inclusive) to max (exclusive)	`range(5)`	0, 1, 2, 3, 4
`range(min, max)`	Range from min (inclusive) to max (exclusive)	`range(3, 7)`	3, 4, 5, 6

`for` loop syntax

Syntax:

```
for variable in range (start, stop):     } header
        statement                        ⎫
        statement                        ⎬ body
        ...                              ⎪
        statement                        ⎭
```

- Set the **loop variable** equal to the **start** value
- Repeat the following:
  - Check if the **variable** is **less than** the **stop**. If not, stop
  - Execute the **statements**
  - Increase the variable's value by 1

```
for i in range(1, 6):     # repeat 5 times
        bake_cookies()
```

Repetition over a range

```
print("1 squared = ", 1*1)
print("2 squared = ", 2*2)
print("3 squared = ", 3*3)
print("4 squared = ", 4*4)
print("5 squared = ", 5*5)
print("6 squared = ", 6*6)
```

- Let’s use a `for` loop ...

```
for i in range(1, 7):
      print(i, "squared = ", i*i)
```