TCSS 142 – Introduction to Programming

Autumn 2014
Day 12

Day 12 Overview

- Programming Assignment 1
- Global variables
- Value returning functions
- Built-in libraries
- Strings

Programming A 1

- Questions?

Global Variables

- Yes, they exist.
- No, do not use them.
- Programmer defined global constants – NOT in Python!!!

Scope

```
def f(z):                   # z is a local name
    print(z)
    print(s)                # global s

s = "And me?"               # global s
f("I like Python")
print(s)                    # global s
```

Scope

```
def f(z):                   # z is a local name
    print(z)
    s = "Me too."           # s is a local name
    print(s)                # shadows the global one

s = "And me?"               # global s
f("I like Python")
print(s)                    # global s
```

Scope

```
def f(z):                   # z is a local name
    print(z)
    global s
    s = "Me too."           # s is global
    print(s)                # global s

s = "And me?"               # global s
f("I like Python")
print(s)                    # global s
```

Using Built-in Functions

- Some built-in functions require import statement, some don't.
- Python docs – the ultimate guide on available built-ins
  - https://docs.python.org/3/library/index.html
- When a function doesn't require the dot notation, we will call it a function; otherwise, we will call it a method

Calling `math` methods

```
import math
math.functionName(parameters)
```

- The `math` methods do not print to the console.
  - Each methods produces ("returns") a numeric result.
  - The results are used as expressions (printed, stored, etc.).

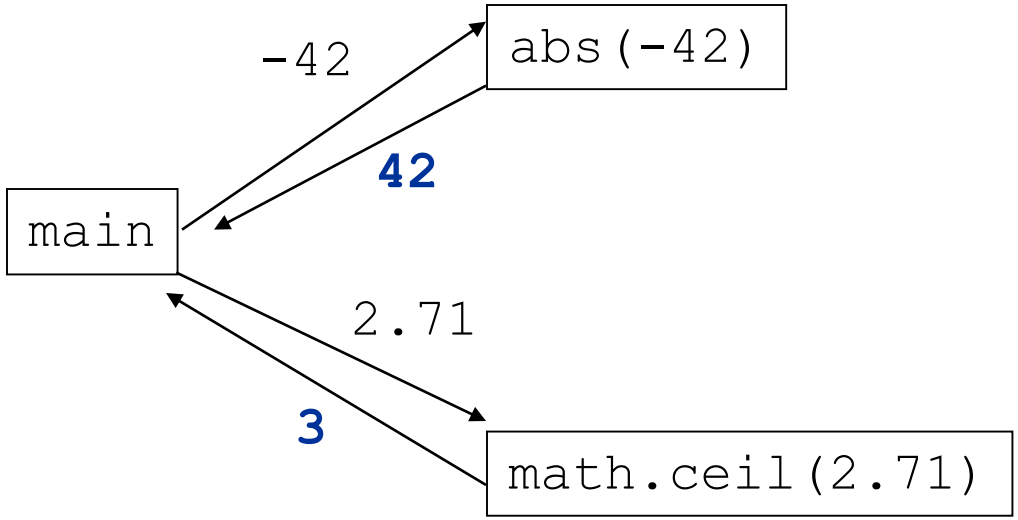
- Examples:

```
squareRoot = math.sqrt(121.0)
print(squareRoot)                    // 11.0

power = math.pow(2, 4)
print(power)                         // 32.0
```

Value-returning Functions

- **return**: To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information *in* from the caller to the method.
    - Return values send information *out* from a method to its caller.
      - A call to the method can be used as part of an expression.



`math` constants

- The `math` module defines variables `pi` and `e`, which are assigned the mathematical values for *pi* and *e*
  - Can be used in equations that require these values, to get more accurate results

- Variables must also be called using the dot notation
  - Example:

```
circle_area = math.pi * radius**2
```

`string` methods

- Some Boolean returning methods

Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Example

```
myString = '1234'

if myString.isdigit():
     print('I am an integer')
     num = int(myString)
else:
     print('Sorry…')
```

`string` methods

- Some return a modified copy of a string

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

Example

- Methods like `lower` and `strip` build and return a new string, rather than modifying the current string.

```
s = "lil bow wow"
print(s.upper())
println(s)                    // lil bow wow
```

- To modify a variable's value, you must reassign it:

```
s = "lil bow wow";
s = s.upper()
print(s)                      // LIL BOW WOW
```

`string` methods

- Some help with searching and replacing

Table 8-3 Search and replace methods

Method	Description
`endswith(substring)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(substring)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns –1.
`replace(old, new)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(substring)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

Exercise

```
str1 = "Gandalf the Gray"
str2 = str1.replace("Gray", "White")

str3 = "str1".replace("r", "range")
str4 = 'oo'
print(str1.replace('a', str4))
```

Returning a Value

```
def name(parameters):
    statements
    ...
    return expression
```

- Example:

```
// Returns the slope of the line between the given points.
def slope(x1, y1, x2, y2) :
    dy = y2 - y1
    dx = x2 - x1
    return dy / dx          # could be an expression, a variable
                            # or a literal constant
```

- `slope(1, 3, 5, 11)` returns `2.0`

Common error: Not Storing

- Many students incorrectly think that a `return` statement sends a variable's name back to the calling method.

```
def caller():
    slope(0, 0, 6, 3)
    print("The slope is " , result)   # ERROR:
                                      # result not defined


def slope(x1, y1, x2, y2) :
    dy = y2 - y1
    dx = x2 - x1
    result = dy / dx
    return result

caller()
```

Common error: wrong return

- Download `returntest.py` from `Canvas` and run.
- Fix the code.
- Delete return statement – what happens now?
- Adjust the slope code in the following fashion – what happens?

```
def slope(x1, y1, x2, y2) :
    dy = y2 - y1
    dx = x2 - x1
    return result
    result = dy / dx
    print("testing return")
```

Returning a Value

- The expression in the `return` statement can be a value, a variable name, complex expression, such as a sum of two variables or the result of another value-returning function.

- In Python, a function can return multiple values
  - Specified after the `return` statement separated by commas
    - Format: `return` *`expression1, expression2`*
  - In a call, you need a separate variable on the left side of the = operator to receive each returned value
    - `var1, var2 = someFunc()`

Exercises

- Download `transposition.py` and save as `transposeFunctions.py`. Then, make the following changes:
  - Move appropriate code to function `main` to handle input and output
  - Move the rest of the code to function `encrypt` that takes plain text as argument and returns the encoded version

- Find `params.py` and save as `circleFunctions.py`. Then, make the following changes:
  - Move appropriate code to function `main` to handle input and output
  - Move the rest of the code to function `calculate` that takes the `choice` and `radius` as arguments and returns the numerical `result` of the calculation, as well a string indicating the calculation type
  - Use `math.pi` instead of variable `pi`
  - User `pow` instead of `radius**2`

Type `boolean`

- **`boolean`**: A logical type whose values are `True` and `False`.

```
minor     = (age < 21)
isProf    = name.startswith("Prof")
lovesCSS  = true

// allow only CSS-loving students over 21
if minor or isProf or  not lovesCSS:
    print("Can't enter the club!")
```

Using `boolean`

- Why is type `boolean` useful?
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it
  - Makes code more readable
  - Can pass around the result of a logical test (as param/return)

```
goodAge    = age >= 21 and age < 29
goodHeight = height >= 78 and height < 84
rich       = salary >= 100000.0

if (goodAge and goodHeight) or rich:
    print("Okay, let's go out!")
else:
    print("It's not you, it's me...")
```

Returning Boolean

- Methods that return Boolean often have an `if/else` that returns `true` or `false`:

```
def bothOdd(n1, n2):
    if n1 % 2 != 0 and n2 % 2 != 0:
        return true
    else
        return false
```

- Calls to methods returning Boolean can be used as tests:

```
if bothOdd(57, 20):
    ...
```

"Boolean Zen"

- Students new to Boolean often test if a result is `true`:

```
if bothOdd(57, 20) == true:      // bad style
    ...
```

- But this is unnecessary and redundant. Preferred:

```
if bothOdd(57, 20):              // good style
    ...
```

- A similar pattern can be used for a `false` test:

```
if bothOdd(57, 20) == false:     // bad style
if not bothOdd(57, 20):          // good style
```

Returning Boolean

- Observation: The `if/else` is unnecessary in `bothOdd`

```
def bothOdd(n1, n2):
    return n1 % 2 != 0 and n2 % 2 != 0
```

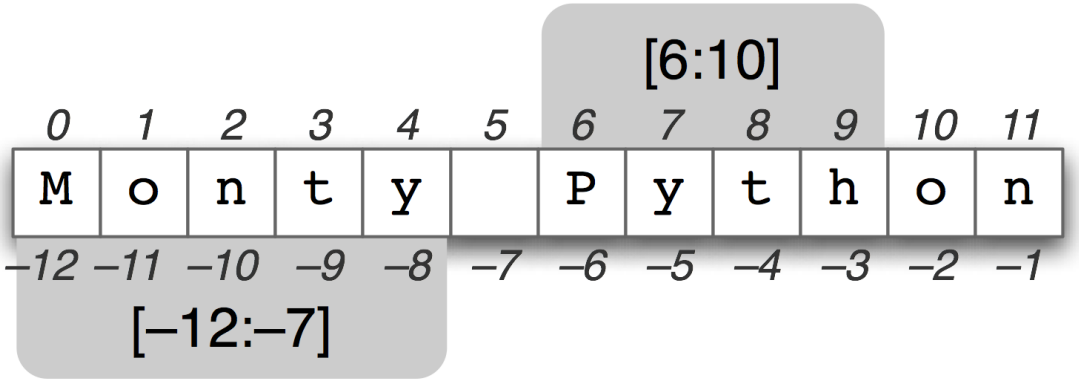
Strings Summary

- What have we learned so far?

More on Strings

- Characters of a string are numbered with *indexes*:

```
phrase = "Monty Python"
```



  - First character's index : 0 or -length
  - Last character's index : length-1 or -1

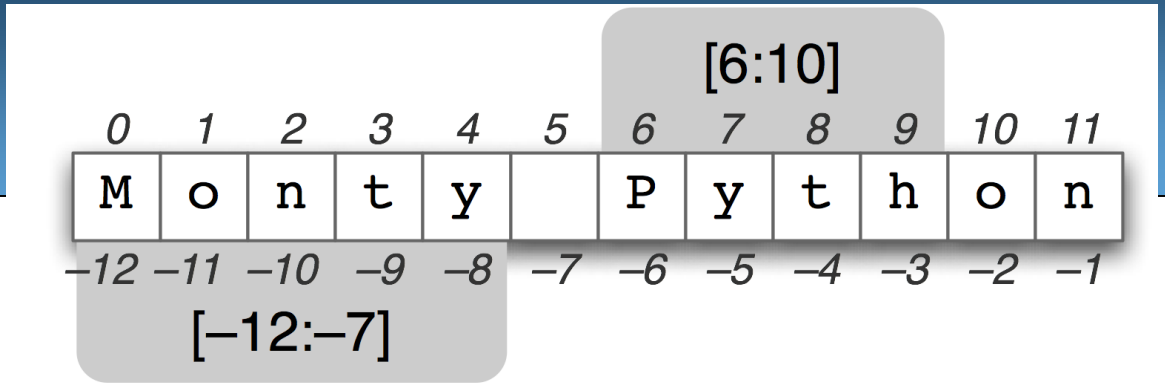
Index

- When we want to get a particular character from a string we use [ ] operator.

```
>>> phrase = "Monty Python"
>>> first1 = phrase[0]
>>> first2 = phrase[-??]
>>> whatAmI = "Monty Python"[6]
>>> print (phrase[20])
>>> len(phrase)
>>> phrase[0] = 'm'
```

Slicing

- Slicing selects a range of characters from a string and returns them as a string
  - uses `[:]` operator
  - format `stringVar[start:end]`
  - `start` – inclusive, `end` – exclusive
  - if start missing, then Python uses a zero, e.g. [: 10] means extract `0 to 9`
  - if end missing, uses string length, e.g. `[5 : ]` means extract `5 to length-1`




```
>>> phrase[0:5]
??
>>> word = phrase[6:12]
>>> word
??
>>> phrase
>>> phrase[:5]
???
>>> phrase[6:]
???
>>> phrase[:]
???
>>> for i in range(len(phrase)+1):
        print(phrase[0:i])
```

Repetition

- To repeat a part of a string, use *

```
>>> 'bon' * 2
bonbon
>>> candy = ("bon" * 2) + 's'
>>> candy
bonbons
```

in and not in

- There are two more string operators
  `in` – determines if one strings contains another
  `not in` – determines if one string does not contain another

```
>>> 'bc' in "Monty Python"
False
>>> 'bc' in phrase
False
>>> 'bc' not in phrase
True
>>> for ch in phrase:
      print(ch)
```

Exercise

- Given the following code, what indexes must be used instead of `a` and `b` to produce the new string with the value `SCORE` ? What indexes to produce `fouryears` ?

```
quote = "Four score and seven years ago"
expr1 = quote[a : b].upper()          #  "SCORE"
expr2 = quote.lower()[a : b] + quote[a : b]
          # "fouryears"
```

Exercise

At the command prompt, declare these variables

```
str1 = "Frodo Baggins"
str2 = "Gandalf the GRAY"
```

And then evaluate the following expressions:

```
len(str1)
str1[7]
str1.find('B')
str1.lower().find('B')
print(str1)
str1[ : 4]
str1 = str1.upper()
print(str1)
```

```
str2[0]
str2.upper()
str2[3:14]
str2[12:]
str2[12:20]
print(str2)
```

Gangsta Name (again)

- Let's redo `gangstaname.py` now that we know slicing
  - step 1 – download and examine
  - step 2 – save as `gangstaslice.py` and redo

Transposition Cipher

	0123..
Original:	It was a dark and stormy night
Even/Odd:	It_was_a_dark_and_stormy_night
Message:	twsandr_n_tryngtI_a__akadsom_ih

Transposition Decryption

- Algorithm
  - Split the message into half
  - Place the first half characters into odd positions of a new string
  - Place the second half characters into even positions of a new string

Typical String Ops

- Remove duplicate letters

```
def removeDupes(myString):
    newStr = ""
    for ch in myString:
        if ch not in newStr:
            newStr = newStr + ch
    return newStr
```

Typical String Ops

- Remove certain letters (as specified in the second param) from the first string

```
def removeMatches(myString, removeString):
    newStr = ""
    for ch in myString:
        if ch not in removeString:
            newStr = newStr + ch
    return newStr
```

Last Slide ☺

- Read chapter 6.1 – 6.3 and complete the quiz by the next class meeting on Tuesday.

- Class ends at 17:10