GUTS Introduction to Programming Workshop
Handout

What is Programming?

Programming is the process of telling a computer what to do through a sequence of commands which the computer executes one after the other.

Expressions and Variables

An **expression** in Python is essentially the building block of your program. Anything that you can manipulate and use is an expression. Here are some examples:

- `56`
- `2 + 3`
- `(5 - 1) * (12 - 4)`
- `“Some text”`
- `“I am attending ” + “a GUTS programming workshop.”`
- `19.32`
- `True`
- `False`
- `[12, 3, -4, 5]`
- `students[2]`

Expressions have a particular **data type**. The main data types in Python are:

- **Number**, this is really representing two types: ints and floats, but you don’t need to worry about this now
- **String** represents pieces of text
- **Boolean** represents values of *`True`* or *`False`*
- **List** represents any ordered collection of items

A **variable** is something which has a particular data type and some value. They are used to store information to be referenced and manipulated in your program. In Python, we define variables by simply giving them a name and a value, like this:

- `x = 12`
- `eventName = “Introduction to Programming Workshop”`

- `y = 2.5`
- `z = x + y`
- `message = “I am attending” + eventName`

With the key thing here is knowing that this is called **assignment** and that it uses the **assignment operator**, =.

Notice how the right side of the assignment operator can be any kind of expression.

Printing

We can print out something to the output using the **print()** function that’s built into Python. Here’s an example:

```
print(“Hello World!”)

x = 10
print(“The value of x is ” + x)
```

You can print pretty much any expression in Python, and we’ll be using it extensively during the workshop.

User Input

We can take in user input using the **`input()`** function. For example:

```
name = input(“What is your name?”)
```

will store whatever the user inputs into the variable name. The parameter in the brackets of the input function represents what to display as the prompt for the input. Try it yourself!

We can use this to build more complex programs which depend on the input of a user.

The `input()` function will always give you a string type. What if you want to get number input? For example you need to ask the user about their age. To do this, you need to convert from a string to a number. We do this using either **`int()`** or **`float()`** and inside that we use input. This is called **casting**. For example:

```
    age = int(input(“How old are you?”))
```

This is however prone to failure, because how can we truly enforce the user to input a number. And if they don’t enter a number, we will try to convert it and this will fail. We will not be covering how to handle this here but it’s something to keep in mind.

`if` Statements

An `if` statement lets you specify code that executes only if a certain condition is true. **Conditions** are represented by the Boolean data type. Examples of conditions include:

- `x > 50`
- `y <= 15`
- `name == “John”`
- `x + y >= z`
- `a > 5 and b < 5`
- `age > 18 or age < 15`

You have operators like >, <, >=, <=, == which go between other expressions. Notice that the result of all of these operations is an expression of a Boolean type. It is also important to notice the distinction between == and =. The == operator is used for comparison and tells you whether or not two expressions are equal.The = operator is used for assignment as explained above.

Conditions are used in `if` statements in this syntax:

```
    if <condition>:
        <code to run>
```

<u>A note about indentation in Python</u>

In Python, line indentation matters! Consecutive lines that are on the same level on indentation make up a **code block**. Take for example this short program:

```
x = 5
if x > 10:
     print(“Runs only when x is greater than 10.”)
print(“This will always run.”)
```

`while` Loops

Say you need to run something multiple times. For instance you want to print this statement 10 times:

```
print(“This will run 10 times.”)
```

Well, you can always copy and paste it 10 times, but that’s just not very good. Because you could always change your mind and want to print it 1000 times! So we use loops for this. There are a few types of loops but one of the simplest is the `while` loop:

```
while <condition>:
     <run code multiple times>
```

The `while` loop takes a condition just like the `if` statement and checks it. If it’s true, the code block runs. At the end of the code block, it checks the condition again and runs the block again if it’s true, and so on. Notice that it’s the responsibility of the code block to ensure that the `while` loop eventually terminates.

So we can write our example program that prints something out a certain number of times like so:

```
x = 0
while x < 10:
     print(“This will print many times.”)
     x = x + 1
```

Let’s analyse this! We start off with an `x` set to 0 which we use as our counter. We then start the while loop. The condition is `x < 10` which means that the while loop will run until this condition is no longer satisfied. At the start `x` is obviously less than 10 so the loop runs and we see the statement printing. We then run `x = x + 1`, which simply adds 1 to `x`, making it 1 now. This is the end of the code block and the loop restarts, checking x yet again. It is now 1 which is still less than 10 so the loop runs. This keeps going until after the statement prints 10 times, the value of `x` is now 10 and we move on from the while loop.

Breaking from a loop

Sometimes we want to force exit from a loop without going back to the initial condition. We can do this using the `break` keyword. This will cause the loop to immediately stop and move on. Consider this program for instance:

```
x = 0
while True:
      if x == 5:
            break
      print("The value of x is " + x)
      x = x + 1
```

First off, notice the use of `True` as a condition. This means that the loop will essentially run forever, or at least until a break happens. This loop checks prints out `x` and increases `x` everytime. However, as soon as `x` is equal to 5, it will break and exit. Writing loops like can be very useful in some situations.

Lists

Lists are an important data type which allow you to represent collections of items. For example, say you want to keep track of the names of employees in a company, in Python it would look like this:

```
employees = ["John Smith", "Mona Simpson", "Robert Blake", "Fiona
Clarkson"]
```

There are some operations we should be familiar with regarding lists. You can access elements in a list by using an **index**. The index is specified inside square brackets after the list variable's name. Lists are indexed starting 0, which means that the first element is really zeroth element. For instance the first employee in the above example would be given by:

```
    employees[0]
```

Go ahead and try this! Try accessing other employees as well and printing them. Now say you need to add a new employee to your list. We use the **`append()`** function to do this.

```
    employees.append("Richard Hendricks")
```

But we can also do this: `employees = employees + "Richard Hendricks"` as an alternative way to add elements to lists.

We may at times need to get the length of a list. We can do this using the **`len()`** function:

```
    len(employees)
```

Lastly we might want to remove items from a list, which we can do using the **`remove()`** function:

```
    employees.remove("John Smith")
```

`for` loops

`for` loops are another type of loop which is quite useful when dealing with lists. It allows us to easily go over items in a list and do something with them. They are used like this:

```
    for <item> in <collection to go over>:
        <do something for each item>
```

Take for example the employees list we made above. Say we would like to print out all the employee names in list. We can do this like so:

```
for employee in employees:
      print(“This employee’s name is ” + employee)
```

This goes through the employee list and executes the print statement appropriately. Notice that there’s another variable that we refer to inside the for loop, `employee`. We define the name of this variable in the first line of the for loop and it essentially acts as pointer to the current employee for which the code is currently running. This is best understood with examples, so do follow along.

Exercises

- Write a program that ask the user for three numbers then prints out their product, and then does it again until the user input “quit”.
- Write a program that comes up with a random number between 1 and 20 (ask a tutor about how to do this) and then asks the user to guess it. If they guess it right they’re congratulated and it starts again. If they guess it wrong they’re told whether it’s higher or lower.
- Given a number, construct a **list** of the numbers from the negative of that number to that number and print it.
- Given a list of items, print out the items in reverse.
- Given a list of numbers, go through each number and print out its square.

Resources

There’s a good book about Python called **How to think like a Computer Scientist** that is available online for free. Check it out if you’d like to learn more: http://interactivepython.org/runestone/static/thinkcspy/index.html.

End

Thank you!

Since this was a test run of this kind of workshop, we would very much appreciate your feedback to help improve this for next semester: https://goo.gl/ErcgCf.