Variables and Types

CHAPTER 3

Objectives

To understand the concept of data types.

To be familiar with the basic numeric data types in Python.

To understand the fundamental principles of how numbers are represented on a computer.

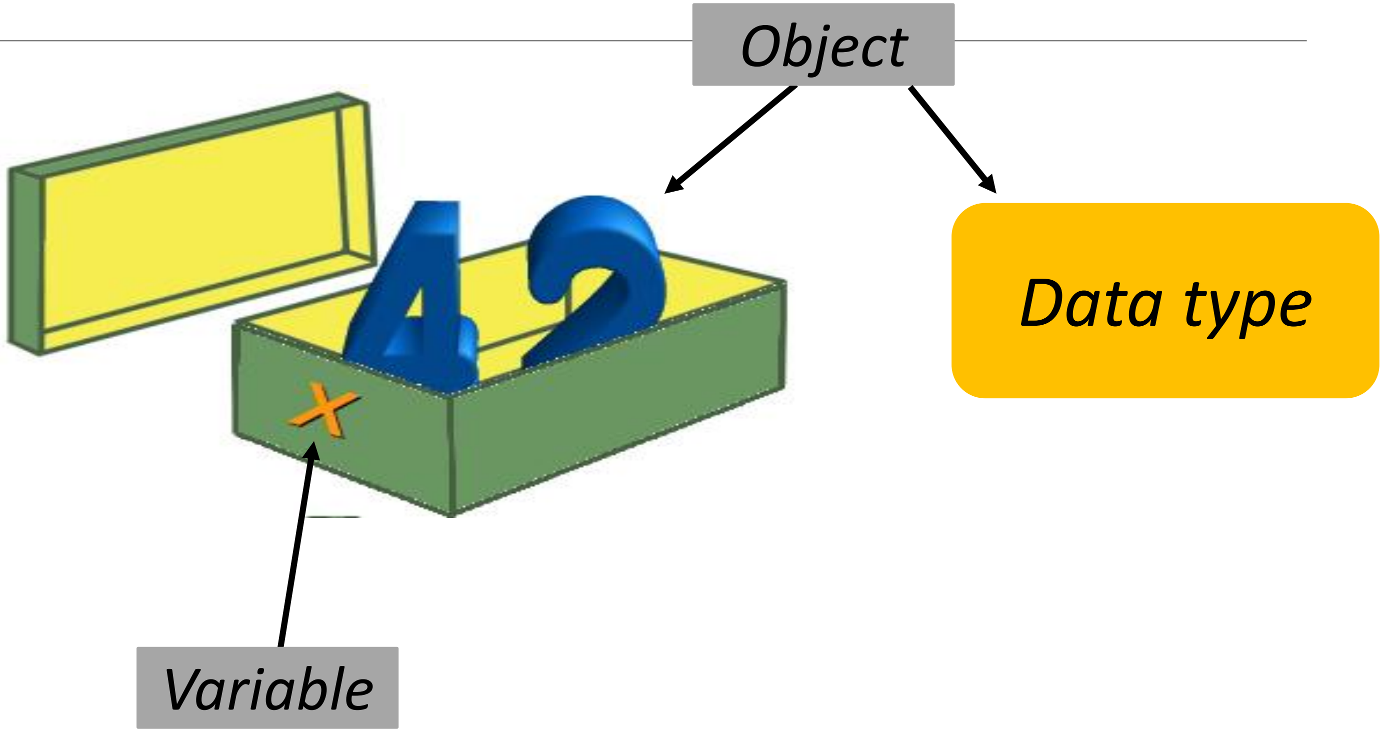
Objectives (cont.)

To be able to use the Python math library.

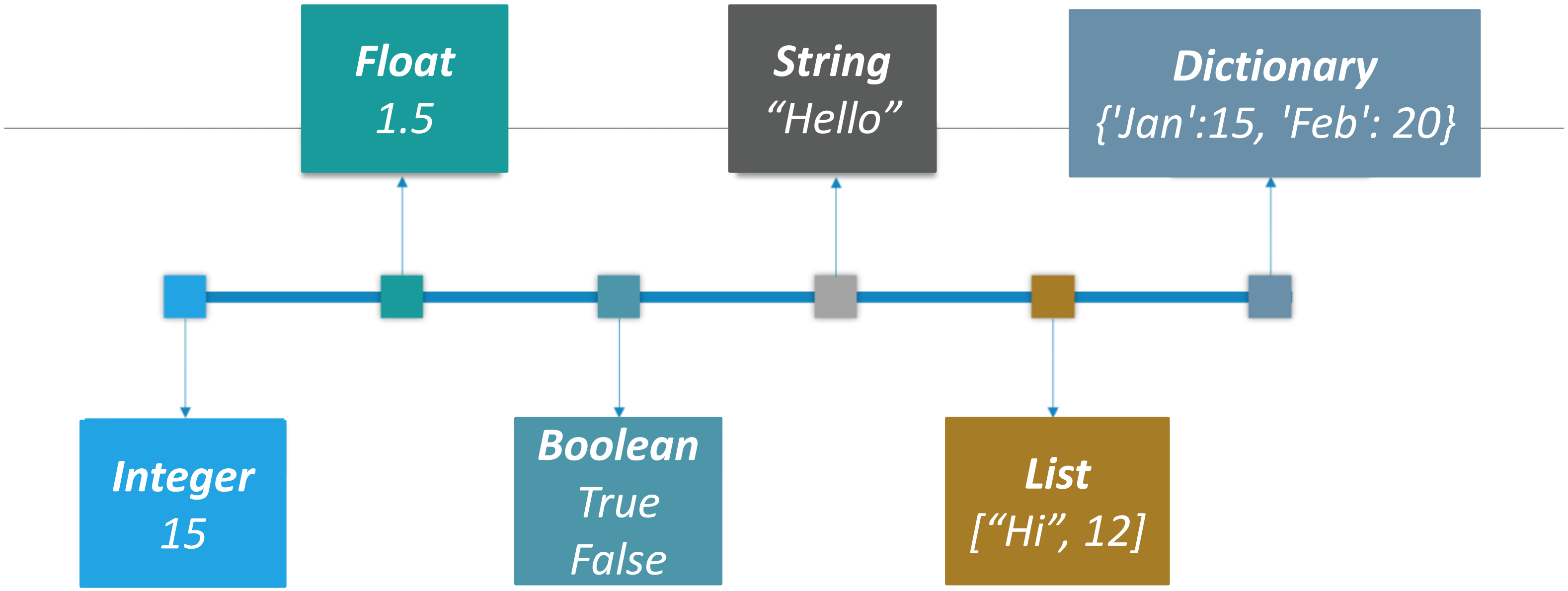
To understand the accumulator program pattern.

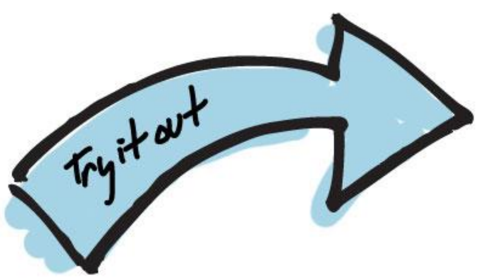
To be able to read and write programs that process numerical data.

Python Variable



Data Types






```
>>> var1=31
>>> var2=16
>>> var3=4.5
>>> var4=3.0
>>> var5=True
>>> var6=False
>>> var7="This is a string"
>>> var8="12345"
>>> var9=[1,3,5,7]
>>> var10=[1,3,5,7,"Odd"]
>>> var11={"Monday":17,"Tuesday":18,"Wednesday":19}
>>> var12={1:"One", 2:"Two", 3:"Three"}
>>> type(var1)
<class 'int'>
>>> type(var3)
<class 'float'>
```

Numeric Data Types

The information that is stored and manipulated by computers programs is referred to as *data*.

There are two different kinds of numbers!

- (5, 4, 3, 6) are whole numbers – they don't have a fractional part
- (.25, .10, .05, .01) are decimal fractions

Numeric Data Types

- Inside the computer, whole numbers and decimal fractions are represented quite differently!
- We say that decimal fractions and whole numbers are two different *data types*.

The data type of an object determines what values it can have and what operations can be performed on it.

Numeric Data Types

Whole numbers are represented using the *integer* (*int* for short) data type.

These values can be positive or negative whole numbers.

Eg: 1, -1, 0, 2, -13

Numeric Data Types

Numbers that can have fractional parts are represented as *floating point* (or *float*) values.

How can we tell which is which?

- A numeric literal without a decimal point produces an int value
- A literal that has a decimal point is represented by a float (even if the fractional part is 0)

Data Type Conversion

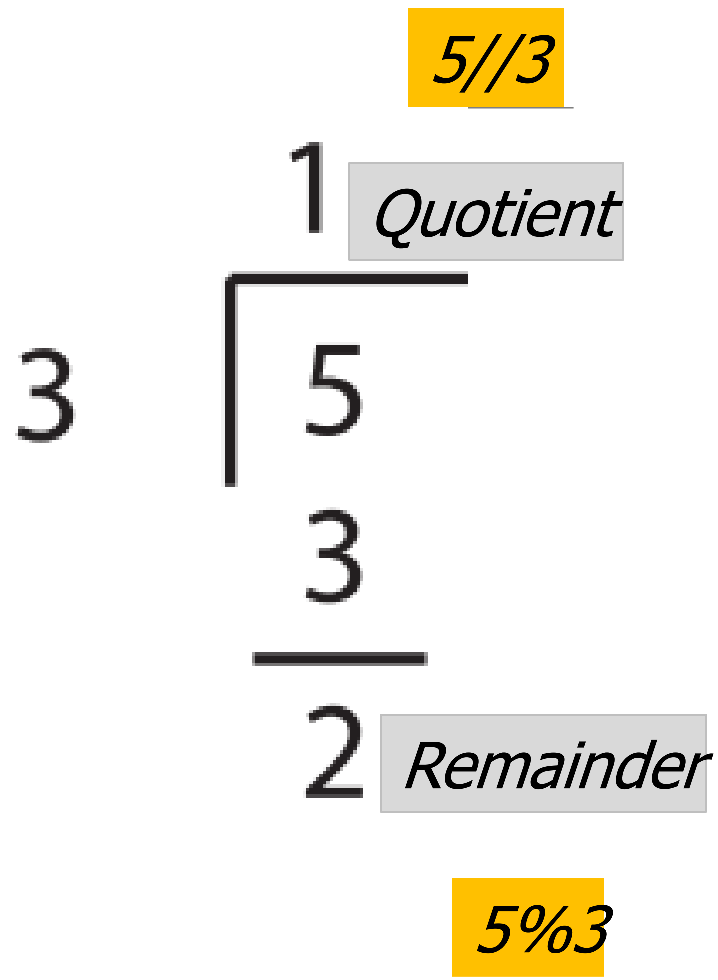
Data type determines

- What is stored
- What kind of operations

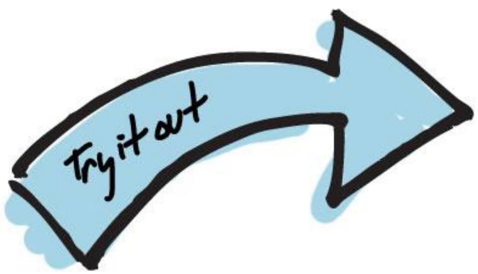
```
>>> var1=31
>>> varV1=str(var1)
>>> varV2=float(var1)
>>> print(varV1)
31
>>> print(varV2)
31.0
>>> type(var1)
<class 'int'>
>>> type(varV1)
<class 'str'>
>>> type(varV2)
<class 'float'>
```

Operations

Symbols	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Quotient/floor division
%	Remainder/modulus
**	Exponentiation



Operations




```
>>> 30*3
90
>>> 30/3
10.0
>>> 30//3
10
>>> 30%3
0
>>> 30**3
27000
>>> 
```

Order of Evaluation

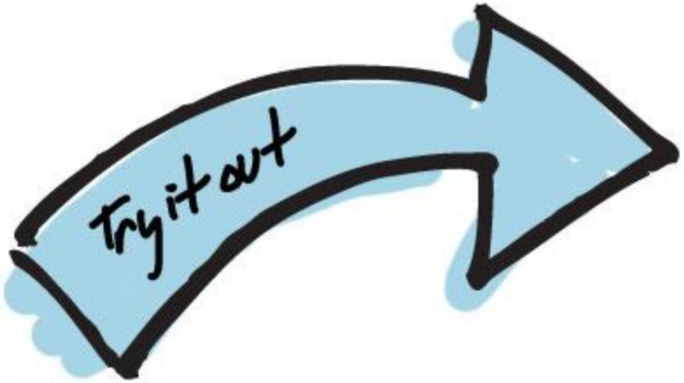
Bracket
( )

Exponential
****

Multiplication, division, remainder, quotient
****, / , %, //***

Addition, Subtraction
+ , -

Operations



2 + 3 – 4

*4 / 2 * 5*

*2 + 3 * 5*

*(2 + 3) * 5*

*2 + 3 * 5**2*

*2 + 3 * 5**2 – 1*

-4 + 2

Operations on Different Data Types

Integer	&	Integer	→	Integer Float (division)
Integer	&	Float	→	Float
Integer	&	String	→	String (multiplication)
String	&	String	→	String

Example

```
>>> var1=3
>>> var2=4.5
>>> var3="This is a string"
>>> var1*var2
13.5
>>> var1*var3
'This is a stringThis is a stringThis is a string'
>>> var2*var3
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    var2*var3
TypeError: can't multiply sequence by non-int of type 'float'
```

Type Conversions

We know that combining an int with an int produces an int, and combining a float with a float produces a float.

What happens when you mix an int and float in an expression?
x = 5.0 + 2

What do you think should happen?

Type Conversions

For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer addition, or convert 2 to 2.0 and do a floating point addition.

Converting a float to an int will lose information

Ints can be converted to floats by adding ".0"

Type Conversion

In *mixed-typed expressions* Python will convert ints to floats.

Sometimes we want to control the type conversion. This is called *explicit typing*.

Type Conversions

```
>>> float(22//5)
4.0
>>> int(4.5)
4
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3)
3
```

Using the Math Library

Besides (+, -, *, /, //, **, %, abs), we have lots of other math functions available in a *math library*.

A *library* is a module with some useful definitions/functions.

Using the Math Library

Let's write a program to compute the roots of a quadratic equation!

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The only part of this we don't know how to do is find a square root… but it's in the math library!

Using the Math Library

To use a library, we need to make sure this line is in our program:

```
import math
```

Importing a library makes whatever functions defined within it available to the program.

Using the Math Library

To access the sqrt library routine, we need to access it as `math.sqrt(x)`.

Using this dot notation tells Python to use the sqrt function found in the math library module.

To calculate the root, you can do
discRoot = math.sqrt(b*b – 4*a*c)

Using the Math Library

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )

main()
```

Using the Math Library

This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 3, 4, -1

The solutions are: 0.215250437022 -1.54858377035

Accumulating Results: Factorial

Say you are waiting in a line with five other people. How many ways are there to arrange the six people?

720 -- 720 is the factorial of 6 (abbreviated 6!)

Factorial is defined as:
n! = n(n-1)(n-2)…(1)

So, 6! = 6*5*4*3*2*1 = 720

Accumulating Results: Factorial

How we could we write a program to do this?

Input number to take factorial of, n
Compute factorial of n, fact
Output fact

Accumulating Results: Factorial

How did we calculate 6!?

6*5 = 30

Take that 30, and 30 * 4 = 120

Take that 120, and 120 * 3 = 360

Take that 360, and 360 * 2 = 720

Take that 720, and 720 * 1 = 720

Accumulating Results: Factorial

What's really going on?

We're doing repeated multiplications, and we're keeping track of the running product.

This algorithm is known as an *accumulator*, because we're building up or *accumulating* the answer in a variable, known as the *accumulator variable*.

Accumulating Results: Factorial

The general form of an accumulator algorithm looks like this:

Initialize the accumulator variable

Loop until final result is reached

Update the value of accumulator variable

Accumulating Results: Factorial

It looks like we'll need a loop!

```
fact = 1

for factor in [6, 5, 4, 3, 2, 1]:
    fact = fact * factor
```

Let's trace through it to verify that this works!

Accumulating Results: Factorial

Why did we need to initialize fact to 1? There are a couple reasons…

- Each time through the loop, the previous value of fact is used to calculate the next value of fact. By doing the initialization, you know fact will have a value the first time through.
- If you use fact without assigning it a value, what does Python do?

Accumulating Results: Factorial

Since multiplication is associative and commutative, we can rewrite our program as:

```
fact = 1
for factor in [2, 3, 4, 5, 6]:
fact = fact * factor
```

Great! But what if we want to find the factorial of some other number??

Accumulating Results: Factorial

What does *range(n)* return?
0, 1, 2, 3, …, n-1

range has another optional parameter! *range(start, n)* returns
start, start + 1, …, n-1

But wait! There's more! *range(start, n, step)* returns
start, start+step, …, n-1

list(<sequence>) to make a list

Accumulating Results: Factorial

Let's try some examples!

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
```

Accumulating Results: Factorial

Using this *range* statement, we can do the range for our loop a couple different ways.

- We can count up from 2 to n:
  range(2, n+1)
  (Why did we have to use n+1?)
- We can count down from n to 2:
  range(n, 1, -1)

Accumulating Results: Factorial

Our completed factorial program:

```
# factorial.py
#    Program to compute the factorial of a number
#    Illustrates for loop with an accumulator

def main():
    n = eval(input("Please enter a whole number: "))
    fact = 1
    for factor in range(n,1,-1):
        fact = fact * factor
    print("The factorial of", n, "is", fact)

main()
```

Accumulating Results: Factorial

What is 100!?

>>> main()

Please enter a whole number: 100

The factorial of 100 is 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000

Wow! That's a pretty big number!

Exercise

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is $10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be convenient to write a computer program to perform the calculation, because then we can experiment with different values for the variables. Here is an algorithm that we can use:

1. *Get the desired future value.*
2. *Get the annual interest rate.*
3. *Get the number of years that the money will sit in the account.*
4. *Calculate the amount that will have to be deposited.*
5. *Display the result of the calculation in step 4.*

The String Data Type

The most common use of personal computers is word processing.

Text is represented in programs by the *string* data type.

A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

Getting a string as input

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

Notice that the input is not `eval`uated. We want to store the typed characters, not to evaluate them as a Python expression.

The String Data Type

We can access the individual characters in a string through *indexing*.

The positions in a string are numbered from the left, starting with 0.

The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.

We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

Indexing returns a string containing a single character from a larger string.

We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.

The String Data Type

Slicing:
<string>[<start>:<end>]

start and end should both be integer

The slice contains the substring beginning at position start and runs up to **but doesn’t include** the position end.

The String Data Type

H	*e*	*l*	*l*	*o*		*B*	*o*	*b*
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The String Data Type

If either expression is missing, then the start or the end of the string are used.

Can we put two strings together into a longer string?

Concatenation "glues" two strings together (+)

Repetition builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

The function *len* will return the length of a string.

```
>>> len("spam")
4
>>> for ch in "Spam!":
        print (ch, end=" ")

S p a m !
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Simple String Processing

Usernames on a computer system

- First initial, first seven characters of last name

```
# get user's first and last names
first = input("Please enter your first name (all lowercase): ")
last = input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
uname = first[0] + last[:7]
```

Simple String Processing

```
>>>
Please enter your first name (all lowercase): john
Please enter your last name (all lowercase): doe
uname =  jdoe

>>>
Please enter your first name (all lowercase): donna
Please enter your last name (all lowercase): rostenkowski
uname =  drostenk
```

Simple String Processing

Another use – converting an int that stands for the month into the three letter abbreviation for that month.

Store all the names in one big string:
"JanFebMarAprMayJunJulAugSepOctNovDec"

Use the month number as an index for slicing this string:
monthAbbrev = months[pos:pos+3]

Simple String Processing

Month	Number	Position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

- *To get the correct position, subtract one from the month number and multiply by three*

Simple String Processing

```
# month.py
#  A program to print the abbreviation of a month, given its number


def main():

   # months is used as a lookup table
   months = "JanFebMarAprMayJunJulAugSepOctNovDec"

   n = eval(input("Enter a month number (1-12): "))

   # compute starting position of month n in months
   pos = (n-1) * 3

   # Grab the appropriate slice from months
   monthAbbrev = months[pos:pos+3]

   # print the result
   print ("The month abbreviation is", monthAbbrev , ".")

main()
```

Simple String Processing

```
>>> main()
Enter a month number (1-12): 1
The month abbreviation is Jan.
>>> main()
Enter a month number (1-12): 12
The month abbreviation is Dec.
```

One weakness – this method only works where the potential outputs all have the same length.

References

Main References

Zelle, J. (2016). Python programming: An Introduction to Computer Science. (3rd ed.). Washington, USA: Franklin, Beedle & Associates Inc,

Punch, W., & Enbody, R. (2016). The Practice of Computing using Python (3rd ed.). Upper Saddle River, NJ: Pearson Education.

Additional References

Gaddis, T. (2015). Starting out with Python (3rd ed.). Essex, England: Pearson Education Limited.