IT@JCU

CP1401 – Module 2 Input, Processing and Output

JAMES COOK UNIVERSITY AUSTRALIA

Learning outcomes – you will be able to:

- Write Python code to implement programs using input, processing and output
- Choose and name variables effectively

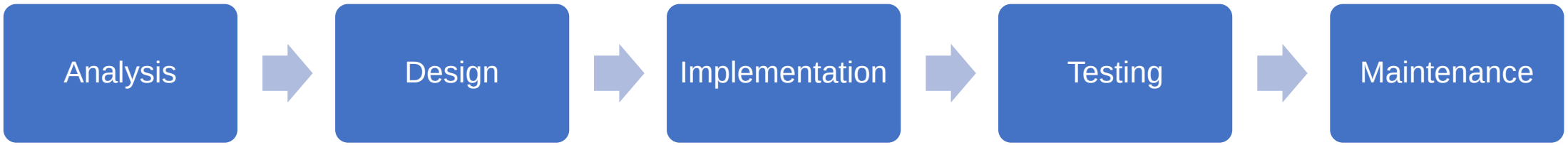
IT@JCU

Revision – Problem Solving

JAMES COOK UNIVERSITY AUSTRALIA

Designing a Program

- Programs must be designed (planned) before they are written
- Program development cycle:



Designing a Program

- **Analysis** is describing **what** a program will do
  - You can’t solve a problem if you don’t understand it

- **Design** is describing **how** the program will do it
  - You can’t implement a solution without knowing how to solve the problem

Designing a Program

- Determine the steps that must be taken to perform the task
  - Break down required tasks into a series of steps - **decomposition**
  - Create an algorithm, listing logical steps that must be taken
- **Algorithm**: set of well-defined logical steps that must be taken to perform a task

Algorithm in pseudocode

```
total = 0
count = 0
get age
while age >= 0
    total = total + age
    count = count + 1
    get age
display total
display total / count
```

Pseudocode should be easy to read & understand, and easy to write.
Use the terms and style you see in our guide and examples.

Notice:

- consistent use of names and terms
- indenting to show structure
- specific details about display (UI) are *not* present/needed

Algorithm with input, processing and output

```
get length, width, depth
volume = length * width * depth
display volume
```

Do this now – write these algorithms

1. Write pseudocode to get a user's name and then greet them with it (e.g., "Hello Monty")

2. Write pseudocode to calculate how much a TV streaming service will cost per year based on a monthly subscription cost

Solutions – something like

1.

```
get name
display greeting with name
```

2.

```
get monthly cost
total cost = monthly cost * 12
display total cost
```

Often, programs follow a three-step process

- Receive **input**
  - Input: any data that the program receives while it is running
- Perform some **processing** on the input
  - Example: mathematical calculation
- Produce **output**
  - On the screen, printer, network, file…

```
get monthly cost
total cost = monthly cost * 12
display total cost
```

Let's start writing Python code!

Beginning with **output**...

```
print("Hello world")
```

Use the **print** function to display output

- Function: piece of prewritten code that performs an operation
- Python's `print` function displays text on the screen
- Argument: data given to a function (inside the parentheses)
  - Example: data that is printed to screen
    Here, "Hello world" is the *argument* passed to the print *function*

```
print("Hello world")
```

Strings and String Literals

- `"Hello world"` is a string
- A string is a sequence of characters
- String literal: actual string that appears in code of a program
  - Must be enclosed in single (') or double (") quote marks
  - If you want apostrophes in a string, use double quotes to enclose, like: "Python's a great language"

Use the **input** function to get input

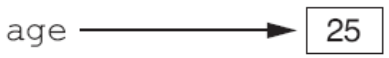
```
input("What is your name? ")
```

- But wait… What do we do with the input?
- Input data (almost always) needs to be stored so it can be used.

We store data in **variables**

- Variable: data value stored in the computer's memory, with a name so we can access it
  - A variable references the value it represents
- Assignment statement: used to create a variable and make it reference data
  - General format is: **`variable = expression`**
    - Example: `age = 25`
    - Assignment operator: the equal sign (=)

Figure 2-4 The age variable references the value 25



Now that we can do input and output

- Here's problem 1 from earlier:

Pseudocode:

```
get name
display greeting with name
```

Python code:

```
name = input("What is your name? ")
print("Hello", name)
```

print can display any number of items

- Items are separated by commas when passed as arguments
- Arguments display in the order they are passed to the function
- By default, items are separated by a space when displayed
  - This can be changed using the *sep* keyword argument

```
print(name, ", your name is ", name, "!", sep="")
```

As soon as we have more than a couple of items, there's an easier way to print, using **f-strings**

```
print(name, ", your name is ", name, "!", sep="")
```

or…

```
print(f"{name}, your name is {name}!")
```

Now let's do some processing, with problem 2

Do this now

- Convert this pseudocode to Python code
  (try your best, based on what you've seen/learned so far)

```
get monthly cost
total cost = monthly cost * 12
display total cost
```

You may have written something like

```
monthly_cost = input("Monthly cost: $")
total = monthly_cost * 12
print("Total cost is $", total, sep="")
```

Looks good, but what happens when we run it?

Monthly cost: $6.99

Total cost is $6.996.996.996.996.996.996.996.996.996.996.99

Every variable has a **type**

Type	Values
`int`	Whole numbers (…, -3, -2, -1, 0, 1, 2, 3, …)
`float`	Real numbers (...,-2.73,-1.0,-0.5,0.0,0.5,3.1415, ...)
`str`	Sequence of characters ('hello world', "answer", …)
`bool`	Boolean value – either True or False
`list`	Collection of objects ([1, 2, 3], ["Python", True, 1]…)
…	

The input function **always** returns a **string**, so…

- Python has built-in functions to convert between data **types**
  - `int(item)` converts item to an int
  - `float(item)` converts item to a float
  - Numeric type conversion only works if item is a valid numeric value, otherwise, throws exception (crashes!)

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

- This is a nested function call: function1(function2(argument))
  - value returned by function2 is passed to function1

You can explicitly convert between types

Initial	Conversion	Meaning
`x = 42`	`float(x)`	Convert the **int** 42 into the **float** 42.0
`s = '42'`	`int(s)`	Convert the **str** '42' into the **int** 42
`x = 42.0`	`str(x)`	Convert the **float** 42.0 into the **str** '42.0'
`x = 4.9`	`int(x)`	Convert the **float** 4.9 into the **int** 4
`s = input()`	`age = int(s)`	Convert the user's **str** `input` to an **int**

- Type conversion is also called "type casting"
- Note! Only the last one actually saves the new value

Try again... using type conversion to store a number

Before (problem):

```
monthly_cost = input("Monthly cost: $")
total = monthly_cost * 12
print("Total cost is $", total, sep="")
```

After (solved):

```
monthly_cost = float(input("Monthly cost: $"))
total = monthly_cost * 12
print("Total cost is $", total, sep="")
```

Here's the same code using an f-string

```
monthly_cost = float(input("Monthly cost: $"))
total = monthly_cost * 12
print(f"Total cost is ${total}")
```

You will often do processing with mathematical operators

- Python understands the standard set of operators:

Operator	Example	Result
Add	`5 + 4`	9
Subtract	`5 - 4`	1
Multiply	`5 * 4`	20
Divide	`5 / 4`	1.25
Integer Divide	`5 // 4`	1
Modulo	`5 % 4`	1
Exponent	`5 ** 4`	625

Understand operator precedence and parentheses

- Higher precedence performed first
- Same precedence operators execute from left to right
- Operations enclosed in parentheses force operations to be performed before others
- Order:
  - Exponentiation (**)
  - Multiplication (*), division (/ or //), and remainder (%)
  - Addition (+) and subtraction (-)

Do this now – write the results of these **expressions**

a. 5 * 2 - 3

b. 3 + 2 ** 6

c. 7 * 3 // 4 - 3 * 2

d. (17.3 / 45.14) ** 2 / 5.1 + 3

You can use Python (console) to check your answers

a. 5 * 2 - 3 = **7**

b. 3 + 2 ** 6 = **67**

c. 7 * 3 // 4 - 3 * 2 = **-1**

d. (17.3 / 45.14) ** 2 / 5.1 + 3 = **3.028800426598512 5**

Do this now – write an algorithm to calculate the day of the week (0-6) for a day in the future, like:

E.g., if today is 1 (Monday) then 14 days from now is also 1/Monday

Sample output:

```
Current day number: 1
Number of days in the future: 14
New day number is 1


Current day number: 5
Number of days in the future: 141
New day number is 6
```

Modulo (mod, %) is useful for repeating cycles

- days of the week: 0-6
- odd, even, odd, even… 0-1
- hours of the day: 0-23
- minutes of the hour: 0-59
- **What else can you think of?**

Modulo gives the remainder of a division

- 15 % 6 = 3
  Because 15 / 6 = 2 remainder **3** (2 * 6 = 12, 15 - 12 = 3)

- If a movie length is 157 mins, this is:
  - 2 hours 157 // 60 = 2
  - and 37 minutes 157 % 60 = 37

- Any number % 2 is either 0 or 1, which is useful:
  - 3 % 2 = 1 odd
  - 314159265 % 2 = 1 odd
  - 1234567890 % 2 = 0 even

So, use % to calculate the day of the week (0-6)

E.g., if today is 1 (Monday) then 14 days from now is also 1/Monday

```
current_day_number = int(input("Current day number: "))
day_increment = int(input("Days in the future: "))
new_day_number = (current_day_number + day_increment) % 7
print("New day number is", new_day_number)
```

Note that we do NOT use `day_number` (generic) and `new_day_number` (specific). Otherwise, we have 2 "day numbers" and have to work out what the generic name refers to (current or future/new?).

`current_day_number` might seem long, but it's unambiguous.

Do this...

- Practise writing small programs that use input, processing and output.
- Try all the different operators
- Keep going until you feel comfortable - like you've "got it"
- Here's one to try now:
  - Input: Ask the user for their age (what type will you need?)
  - Processing: Calculate how old they will be in 10 years
  - Output: Display their age in 10 years

Do this now

- Write an algorithm first, then Python code to calculate a user's net pay after deducting tax. We'll need to know (input) the gross pay and tax rate. (It's simple taxation, no thresholds or different rates)

You may have written something like

```
get gross pay, tax rate
tax amount = gross pay * tax rate
net pay = gross pay - tax amount
display net pay
```

```
gross_pay = float(input("Gross pay: $"))
tax_rate = float(input("Tax rate (e.g. 0.3 is 30%): "))
tax_amount = gross_pay * tax_rate
net_pay = gross_pay - tax_amount
print("Net pay is $", net_pay, sep="")
```

We need good variable names to write good code

- Notice the similar-but-different concepts/values in this problem:

```
get gross pay, tax rate
tax amount = gross pay * tax rate
net pay = gross pay - tax amount
display net pay
```

- We can't use names like "tax" and "pay" because: what's "tax"? Do we mean the rate or the amount?
- So we need to make the distinctions clear
  - `tax_rate, tax_amount`
  - `gross_pay, net_pay`

What if we use poor names?

- Notice the similar-but-different concepts/values in this problem:

```
get pay, tax
tax2 = pay * tax
net pay = pay – tax… wait, tax or tax2, I'm getting confused
display pay
```

- Poor names lead to mistakes. We need to think *clearly* about a problem and a solution.
- If we have names like tax, tax2 and pay, net pay… we can't easily tell the difference, so we easily mistake them.

Here's a strong guideline

- When you have different items with similar names:
  - name them differently so the distinction is clear
  - never name one generally and the other specifically

- Bad:
  - tax, tax2
  - pay, net pay
  - name, first name
  - score, total score
  - thing, thingie, thingo

Python identifier naming rules

- must begin with a letter or underscore _
  - `ab_123` is OK, but `123_ABC` is not.
- may contain letters, digits, and underscores
  - `this_is_a_2nd_identifier_123`
- may be of any length
- upper and lower case letters are different
  - `Length_Of_Rope` is not `length_of_rope`

Follow the naming conventions

- Fully described by PEP8 (and see Google's Python Style Guide)
  - https://www.python.org/dev/peps/pep-0008
  - https://google.github.io/styleguide/pyguide.html

- the standard way for most things named in Python is **lower with under**, lower case with separate words joined by an underscore:
  - `this_is_a_variable`
  - `monthly_cost`
  - `display_report`

Good identifier naming is **very** important for producing quality code

- *100%* of the identifiers in a program should be **meaningful**
  - Variable names, function names, constants…
- Best-practice suggestions:
  - Don't be cute, don’t use abbreviations (What is atm?)
  - Use intention-revealing names
  - Avoid names that imply something that isn't true (including: don't reuse names for different things)
  - Use pronounceable names (modymdhms?)
  - Avoid mental mapping (a = number, b = total…)
  - Make meaningful distinctions (account, account_data, account_details, account_info… what's the difference?)

Example naming

- **distance_in_metres** reveals its intent better than **distance**
- using **d** to map to the concept of distance is dangerous
- **monthly_rainfall** is more pronounceable than **mthly_rfall**
- **game_over** is a poor name to represent that a game is running, since it implies another possible meaning
- It’s standard practice to use names such as **i** and **j** as indices inside loops
  - Do NOT use these for other things (e.g., names) or programmers (maybe you) will mistake them for indices… and create bugs
  - x, y, z etc. are "taken" (mathematics)

Watch out for "Magic Numbers"

- A magic number is an unexplained numeric value that appears in code. Example:

```
amount = balance * 0.069
```

- What is the value 0.069? An interest rate? A fee percentage? Only the person who wrote the code knows for sure… and they might forget.

The problem with Magic Numbers

- It can be difficult to determine the purpose of the number.
- If the magic number is used in multiple places in the program, it takes too much effort to change the number in each location, should the need arise.
  - What if you don't change it in every place?
- You risk making a mistake each time you type the magic number in the code
  - E.g., you intend to type 0.069, but accidentally type .0069.
    This mistake will cause errors that can be very difficult to find.

Make code better by replacing magic numbers with CONSTANTS

- A named constant is a name that represents a value that **does not change** during the program's execution.
- Example - note how we use ALL_CAPS for constants:

```
INTEREST_RATE = 0.069
```

- This creates a named constant, `INTEREST_RATE`, assigned the value 0.069. It can be used instead of the magic number:

```
amount = balance * INTEREST_RATE
```

Advantages of using named constants

- Named constants make code more self-explanatory (self-documenting)
- Named constants make code easier to maintain (change the value assigned to the constant, and the new value takes effect everywhere the constant is used)
- Named constants help prevent typographical errors

Here are some **guidelines** for constants

- Any time you need a literal **more than once**, turn it into a CONSTANT
- If a name (constant) is more helpful than the number (literal)
- Ask yourself: *"What if I wanted to change this later?"*

Here are some **rules** for constants

- If you have a constant, then you MUST use it everywhere the value exists.
- Never name VARIABLES in CAPS – that's misleading

```
# Bad :(
INTEREST_RATE = 0.069
DEPOSIT = float(input("Deposit: "))
balance = balance + DEPOSIT
amount = balance * 0.069
```

Now do these next steps

- Practise writing algorithms and programs that use I, P, O
  - Write similar-but-different programs
- As you practise, get into the habit of good variable naming
- Read chapter 3 of your textbook (decision structures)